
Legion 1.8

Reference Manual

The Legion Group

Department of Computer Science
School of Engineering & Applied Science
University of Virginia
151 Engineer's Way, P.O. Box 400740
Charlottesville, VA 22904-4740
legion@virginia.edu
<http://legion.virginia.edu/>

Copyright © 1993-2001 by the Rector and Visitors of the University of Virginia.

All rights reserved.

Permission is granted to copy and distribute this manual so long as this copyright page accompanies any copies. The Legion system software herein described is intended for research and is available free-of-charge for that purpose. Permission is not granted for distributing the Legion system software outside of your site.

In no event shall the University of Virginia be liable to any party for direct, indirect, special, incidental, or consequential damages arising out of the use of the Legion system software and its documentation.

The University of Virginia specifically disclaims any warranties, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. The software provided hereunder is on an "as is" basis, and the University of Virginia has no obligation to provide maintenance, support, updates, enhancements, or modifications.

This work partially supported by DARPA (Navy) contract # N66001-96-C-8527, DOE grant DE-FD02-96ER25290, DOE contract Sandia LD-9391, Northrup-Grumman (for the DoD HPCMOD/PET program), DOE D459000-16-3C and DARPA (GA) SC H607305A

Before you start

1.0	Introduction	5
1.1	About this manual	5
1.2	Style conventions	5

Legion commands

2.0	Command-line functions	7
2.1	Calls on objects	7
2.2	Calls on class objects	14
2.3	Calls on LegionClass	22
2.4	Calls on file and context objects	26
2.5	Start-up and shutdown functions	38
2.6	Scheduling support	45
2.7	General functions about the state of the system	57
2.8	Security	62
2.9	Application development	73
2.10	Program support	80
3.0	Error codes	115
4.0	Internal tools	117

Manipulating hosts and vaults

5.0	Alternative ways to start and shut down	119
5.1	Starting extra hosts and vaults, command line	119
5.2	Shutting down extra hosts and vaults, command line	120
5.3	Restarting extra hosts and vaults	120
5.4	Using two hosts to start up a system	120

Runtime library examples and interfaces

6.0	Example translations: the simple class	122
6.1	Simple.h and Simple.c	122
6.2	Simple.trans.h and Simple.trans.c	123
6.3	ex1_Simple.c	127

7.0	Interfaces	130
7.1	LegionProgramGraph	130
7.2	LegionInvocationStore	131
7.3	LegionWorkUnit	132
7.4	LegionBuffer	132
7.4.1	<i>Constructors</i>	132
7.4.2	<i>Operations on the associated LegionStorage</i>	133
7.4.3	<i>Operations on the associated LegionPacker</i>	134
7.4.4	<i>Operations on the associated LegionEncryptor</i>	135
7.4.5	<i>Operations on the associated LegionCompressor</i>	135
7.4.6	<i>LegionPackable</i>	135
7.4.7	<i>Other functions</i>	135
7.4.8	<i>Naming and binding: LegionLOID</i>	135
7.5	LegionMessage	138
7.5.1	<i>LegionParameter</i>	138
7.5.2	<i>LegionParameterList</i>	139
7.5.3	<i>LegionComputationTag</i>	140
7.5.4	<i>Other fields</i>	140

Legion Grid library

8.0	Grid library	141
8.1	Legion Library interface	143
8.2	Exceptions	146
8.3	Legion context space	147
8.4	Legion object management	152
8.5	Legion attributes	168
8.6	Interfaces to Legion objects	175
8.7	Legion C++ type manipulation	176
8.8	Legion host object	180
8.9	Vault object	185
8.10	Class object	189
8.11	Legion implicit parameters	191
8.12	Collections	198
9.0	Error messages	202
	Getting help	203
	Index	204

Before you start

1.0 Introduction

1.1 About this manual

This manual provides detailed information about specific parts of the Legion system. It is intended to be a supplement to the other Legion documentation, and new users are advised to start with the Basic User Manual.

There are four Legion manuals, each aimed at a specific type of user, that can be consulted for more information. The others are:

- **Basic User Manual:** introductory information about the system.
- **System Administrator Manual:** information and documentation for administrators of Legion systems, including installing and running Legion, configuring security features, resource management, and managing a Legion system.
- **Developer Manual:** information and documentation for programmers working in Legion, and includes information on languages, libraries, core objects, and implementing new Legion objects.

There are also man pages for all Legion commands, included with the system files, and several on-line tutorials on the Legion web site (<http://legion.virginia.edu>).

1.2 Style conventions

The manuals at times refer to path names in Unix directory space and in Legion context space. To avoid confusion, the following style conventions are used throughout Legion documentation:

- Unix, DOS, and local path names appear in a serif typeface.
- Functions, method names, parameters, flags, command-line utilities (such as `rm`, `cp`, and `legion_ls`), and context path names appear in fixed typeface.

Legion commands

Legion's command-line tools support path-like naming of objects within context space. There are man pages for all of the commands listed here as well as resources on the Legion web site (<http://legion.virginia.edu>). Man page usage is `man <command name>`.¹ Please see your system administrator if you are unsure about using man pages on your system.

There are other sources of information about Legion context space and commands: the Legion web site has on-line tutorials and a quick guide to all commands (see <http://legion.virginia.edu/documentation.html>). You can also type

\$ <command name> -help

to see most commands' full usage.

The following syntax is used to indicate optional and required parameters:

[flag]	Optional. You do not have to include this in order to use the tool.
<argument>	Required. You must include this in order to use the tool.
{<argument 1> <argument 2>}	You must include one of these two arguments in order to use the tool.
+	You can use this argument or flag more than once when running the tool.

Please note that the sample outputs may not match your own. Your system administrator may have customized your system's context space or limited what parts you can see.

In the interests of space, most example LOIDs are shortened.

¹ You can also go to http://legion.virginia.edu/legion_man.html to see on-line versions of the Legion man pages.

2.0 Command-line functions

2.1 Calls on objects

`legion_configure_profile`

Edits information about your user profile, security settings, and fault tolerance settings. It is menu-driven. Hit the <enter> key to return to the previous menu level. When you run this command you'll be given the following options:

1. Edit Profile
2. Security
3. Fault Tolerance

The first option lets you edit your AuthenticationObject to include your e-mail address, name, and company.

The second option changes your security preferences for the message layer. You have the following choices:

1. Off (No encryption/No digest)
2. Protected (Digested messages/Encrypted credentials)
3. Private (Encrypted messages/Encrypted credentials)

These will change the settings for messages passed by your objects. We ***STRONGLY*** recommend that you read section 7.0 in the System Administrator Manual before using these options.

The third edits your fault tolerance settings for SKCC objects. The options are:

1. Edit SKCC Policy (default # backups - per user)
2. Edit SKCC Policy (default # backups - class-wide, class owner only)
3. Edit SKCC Vaults (class owner only)

These affect your SKCC objects (BasicFileClass, ContextClass, ImplementationObject, and UserAuthenticationObject are the default SKCC objects). Please see section 6.6 in the Basic User Manual for information on SKCC. Note that these options will affect previous settings. It uses the following commands to edit your settings:

- `legion_skcc_set_class_vaults` (page 11)
- `legion_skcc_set_defaults` (page 11)
- `legion_set_backup_vaults` (page 18)
- `legion_class_vault_list` (page 46)

Note that you cannot change settings on objects that you do not own or have permission to edit. If you are not the class owner, you can use the first option to set the default number of backup vaults for your SKCC objects.

`legion_exports_interface`

```
{[-c] <context name> | -l <LOID>}
{-w <well-known class type> | -f <function signature>}+
[-debug] [-help]
```

Creates an interface from the list of <well-known class type> and <function signature> arguments specified in the argument list. Sends this interface to the object specified by <LOID> or <context path> in the form of a call to the object mandatory function named `exportsInterface()`. Prints to standard output the return value from the call:

- 1 if the interface of the object contains the entire interface of functions specified by the user
- 0 if any one or more of the functions are not exported by the object
- 1 (without contacting the specified object) if the user creates a malformed argument list

The <well-known class type> argument is a string in the set of well-known strings. The set can be listed with `legion_wellknown_class` (see page 60).

For this tool's purposes, `ClassObject` is a well-known class string. `CommandLineClass` and `BootstrapMetaClass` are not considered well-known classes because they do not have any special member functions, as shown in the examples below.

```
$ legion_exports_interface /class/LegionClass \
-w ClassObject
1
$

$ legion_exports_interface /class/LegionClass \
-w CommandLineObject
"CommandLineObject" is not a well known class. Exiting.
-1
$

$ legion_exports_interface /class/LegionClass \
-w LegionClass -w ClassObject
1
$

$ legion_exports_interface /hosts/BootstrapHost \
-w UnixHostClass
1
$

$ legion_exports_interface /hosts/BootstrapHost \
-w ClassObject
0
$
```

```

$ legion_exports_interface /class/LegionClass \
-f " LegionLOID ping();"
1
$

$ legion_exports_interface /hosts/BootstrapHost \
-f " LegionLOID ping();" -w UnixHostClass
1
$

```

The following options are supported:

-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_get_interface

```
{[-c] <class context name> | -l <class LOID>} [-debug] [-help]
```

Retrieves the interface of a Legion object, named by <class LOID> or <context path>, by calling that object's `getInterface()` member function. Prints the returned interface to stdout.

The following options are supported:

-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_ping

```
{[-c] <object context name> | -l <object LOID>}
[-timeout <seconds>] [-debug] [-help]
```

Calls the object-mandatory `ping()` member function on the object named in <object LOID>. If the command returns, the object exists in an active state and its LOID is displayed.

```

$ legion_ping foo
Pinging: 1.01.66000000.14000000.000001fc0a72...
Returned: 1.01.66000000.14000000.000001fc0a72...
$

```

If the object is not accessible by the tool, the command will return an error.

```

$ legion_ping foo
"foo" does not resolve to a valid loid in my context.

```

The following optional parameters are available:

-timeout <seconds>	The time-out flag specifies a maximum number of seconds to wait for the ping to complete successfully. If the object does not respond to the ping within that amount of time, <code>legion_ping</code> will exit. Please note
---------------------------------	---

that `legion_ping` failing due to a user-specified time-out does *not* necessarily mean that the object is inactive or otherwise unreachable. There is no default time-out setting.

-debug

Catch and print Legion exceptions.

-help

Print command syntax and exit.

`legion_list_attributes`

```
{[-c] <object context name> | -l <object LOID>}
[-L] [<attribute name>] [-debug] [-help]
```

Lists an object's attributes from the command line. Optional parameters do the following:

-L

Lists the LOID of each attribute.

<attribute name>

Specify the attribute to be listed (more than one attribute can be listed).

-debug

Catch and print Legion exceptions.

-help

Print command syntax and exit.

A very simple example of this command is below. An object's attributes can include architecture, operating system information, host or vault compatibility, encryption information, etc.

```
$ legion_list_attributes Foo
Foo:
  (ALL)
Total attributes retrieved 1
  favoritecolors('puce', 'apricot')
$
```

`legion_list_invocations`

```
{[-c] <object context name> | -l <object LOID>} [-debug] [-help]
```

Prints a list of currently pending, started, and recently completed invocations for a given object. The output will include information about each invocation's status, timing, progress, and errors.

The following options are supported:

-debug

Catch and print Legion exceptions.

-help

Print command syntax and exit.

`legion_object_info`

```
{[-c] <object context name> | -l <object LOID>}
[-v] [-debug] [-help]
```

Prints information about a given object. The output (obtained from the object's class) will include the object's host, vault, owner, Object Address, status, etc. Optional parameters do the following:

-v	Print additional details about the specified object (host machine name and context name, OPA, and vault context name).
-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

```
legion_skcc_set_class_vaults
<SKCC class context path>
```

```
{[-a | -d] [-c <vault1 context path> | -l <vault1 LOID>]
 [-a | -d] [-c <vault2 context path> | -l <vault2 LOID>] ...
 [-a | -d] [-c <vaultn context path> | -l <vaultn LOID>]}
```

Set an allowable set of backup vaults on which to store the specified SKCC class's replicated state. This applies to all of the class's instances.

Backup vaults are used to replicate an object's persistent state: the object's persistent state is replicated and stored in multiple vaults when it is deactivated, so that if its main vault crashes, its state is still available. However, the replication policy synchronizes the object's state when the object deactivates: if the main vault crashes while the object is still active, the backup vaults will not have the object's current state. If the object needs to be reactivated, it will use an out-of-date state.

This overwrites any policies set via `legion_class_vault_list` (page 46). If you do not use `legion_skcc_set_class_vaults`, Legion will use the class vault restriction lists as specified with `legion_class_vault_list`. The specified class must belong to the SKCC metaclass. Current members are `BasicFileClass`, `ContextClass`, `ImplementationObject`, and `UserAuthenticationObject`.

You can use `legion_set_backup_vaults` to set vaults for a specific instance (page 18). Note, however, that `legion_set_backup_vaults` will overwrite the policy set by `legion_skcc_set_class_vaults` for that instance.

The following parameters must be used:

-l <SKCC class LOID> or -c <SKCC class context path>	Specify an SKCC class object.
-a	Add the specified vault to the object's list of backup vaults
-d	Delete the specified vault

```
legion_skcc_set_defaults
```

```
{[-l <SKCC class LOID> | -c <SKCC class context path>]
 <number of default vaults>}
```

Specifies default level of state replication for instances of an SKCC class. Legion will now by default attempt to copy the instances' state on the specified number of vaults. You must be the class's owner to set this default.

The specified class must belong to the SKCC metaclass. You must have previously set up the class's backup vaults with the `legion_skcc_set_class_vaults` command (page 11). Currently, `BasicFileClass`, `ContextClass`, `ImplementationObject`, and `UserAuthenticationObject` are SKCC classes.

The following parameters must be used:

-l <SKCC class LOID> or **-c** <SKCC class context path>

Specify an SKCC class object.

<number of default vaults> The maximum number of backup vaults that will hold copies of the SKCC class object's state.

`legion_set_worm`

`[[ -c ] <object context path> | -l <object LOID>]`

This command tells Legion that the specified object has Write-Once Read-Many (WORM) semantics and can be used for caching and replicating purposes.

It should be used in conjunction with `legion_unset_worm` (page 12) and `legion_set_backup_vaults` (page 18).

`legion_unset_worm`

`{ [ -c ] <instance context path> | -l <instance LOID> }`

This command tells Legion that the specified object no longer has Write-Once Read-Many (WORM) semantics.

It should be used in conjunction with `legion_set_worm` (page 12) and `legion_set_backup_vaults` (page 18).

`legion_update_attributes`

`{ [ -c ] <object context path> | -l <object LOID> }`
`[-a <new attribute>]`
`[-d <attribute>]`
`[-r <old attribute> <new attribute>]`
`[-debug] [-help]`

Adds, deletes, and replaces an object's attributes. The attribute(s) takes the form "name('val1...valn')". The attribute description must not contain any spaces or begin with an uppercase letter.

Optional parameters do the following:

-a <new attribute> Add an attribute.

-d <attribute> Delete an attribute.

-r <old attribute> <new attribute>
Replace an attribute.

-debug Catch and print Legion exceptions.

-help Print command syntax and exit.

The following example adds the attribute `color('red,blue')` to object `Foo`, with the `-a` flag.

```
$ legion_update_attributes Foo -a "color('red,blue')"  
legion_update_attributes: Added 1 attributes(s) to  
object  
$
```

Similarly, you can remove attributes, with the `-d` flag. Note, however, that if you must provide all of an attribute's parameters:

```
$ legion_update_attributes Foo -d "color('red')"  
legion_update_attributes: Warning - Deleted 0  
attributes(s) from object instead of 1 specified  
  
$ legion_update_attributes Foo -d "color('red,blue')"  
legion_update_attributes: Deleted 1 attributes(s) from  
object  
$
```

2.2 Calls on class objects

legion_activate_object

```
{[-c] <object context name> | -l <object LOID>}
[-debug] [-help]
```

Activates the object named in <object LOID> or <context path> (i.e., instantiates that object as a process) if it is inert. If the object is already active, the command has no effect.

The following options are supported:

-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_copy_class

```
[-v] [-h]
{[-c <template context> | -l] <template LOID>}
<new class path> [-a <attribute>]*
{-mc [-c <metaclass context> | -l <metaclass LOID>]}
```

Make a copy of a class object. You must use an existing class object as the template. The command creates another class object that has the same metaclass and implementations as the template.

The following parameters are required:

-c <template class path> -l <template class LOID>	The original, existing class object.
<new class path>	The new class object's context path.

The following options are supported:

-v	Run in verbose mode.
-h	Print command syntax and exit.
-a <attribute>	Specify an attribute that should be copied from the template to the new class object. You can use this parameter multiple times to specify more than one attribute.
-mc [-c <metaclass path> -l <metaclass LOID>]	Specify the new class object's metaclass (default is the template's metaclass).

legion_create_object

```
{[-c] <class context name> | -l <class LOID>}
<new object context path>
[-h <host name on which to place new object>]
[-v <vault on which to place new object>]
[-H <context path of preferred host class>]
[-V <context path of preferred vault class>]
[-Ch <context containing list of preferred hosts>]
[-Cv <context containing list of preferred vaults>]
[-d <recorder context path> <debug session name>]
[-debug] [-help]
```

Creates an instance of the class named in <class LOID> or <class context path>. No start-up parameters will be supplied for the class or new object. The following options are supported:

- h** Specify a host for the new object.
- v** Specify a vault for the new object.
- H** Specify the preferred host class's context path.
- V** Specify the context path of the preferred vault.
- Ch** Specify a context which contains a list of the preferred hosts.
- Cv** Specify a context which contains a list of the preferred vaults.
- d** Automatically start a `legion_record` session for the newly created object. This allows you to debug server objects. The object's relevant activity will be recorded by a previously started Legion recorder object, named in <recorder context path>.
- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

If the `-h` flag isn't used, the host is selected by the class. Similarly, the class will choose a vault if the `-v` flag isn't used. Normally, this means that a random host is selected, but some classes may act differently. If the `-Ch` or `-Cv` flag is used, the class will randomly choose a host or vault from the hosts or vaults listed in the specified context. In both cases, the system will not return the LOID of the randomly chosen host. The `legion_host_vault_list` and `legion_vault_host_list` (page 47 and page 55) commands will allow users to limit the placement of a given class's instances (i.e., any instances of class `Foo` can only be placed on hosts X, Y, and Z).

legion_create_object_r

```
{[-c] <class context> | -l <class LOID>} <new object context>
<host name> <host architecture> <$LEGION> <$LEGION_OPR>
<$LEGION_OPA> <binary path> [<user id>]
[-debug] [-help]
```

Create a new object on a specified host with the `rsh`² (remote shell) mechanism. The object will be managed with `rsh`, if the class it is

² If preferred, `ssh` can be used in place of `rsh`.

invoked on is an `rshStartClass`. If this utility is invoked on a normal class, normal object create mechanism will be used, and the object will not be managed by `rsh`. This command is generally used only by the `legion_create_class` and `legion_create_vault` scripts, not by users.

The additional arguments specify information for the `rsh` environment.

The following parameters are required:

-c <class context> or -l <class LOID>	The new object's class.
<instance context path>	The new object's context path.
<host name>	Specifies the host upon which the new object should be placed. Note that this should be a DNS name.
<host architecture>	Specifies the host's architecture.
<\$LEGION>	Specifies the Legion environment variable on the <code>rsh</code> host.
<\$LEGION_OPR>	Specifies <code>LEGION_OPR</code> for host.
<\$LEGION_OPA>	Specifies the OPR address for the object, i.e, a unique directory in which the object will maintain its persistent representation on the remote host.
<binary path>	Binary executable path for the object on the remote host.

Optional parameters do the following:

<user id>	Specifies the appropriate user name on the <code>rsh</code> host.
-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

`legion_deactivate_object`

```
[stay_down] [-debug] [-help]
{[-c] <object context name> | -l <object LOID>}
```

Deactivates the object named in <object LOID> or <context path> (moves it to an inert state) if it is currently active. If the object is already inactive, the command has no effect.

The following options are supported:

-stay_down	Forces object to stay inactive. It can only be reactivated by <code>legion_allow_activation</code> (page 26).
-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_destroy_object

```
{[-c] <object context name> | -l <object LOID>}
[-debug] [-help]
```

Deletes the object named in <context path> or <object LOID>. More specifically, it removes the object's LOID but not its context name (if there is one). If the object is active, the command deactivates the object. In all cases, it deletes the OPR (object persistent representation) associated with the object.

This command will not remove any context name associated with the object: you must use the `legion_rm` command to remove the object's name(s) or you will get binding errors. (You can use `legion_ls -A` to check for multiple context names.)

The following options are supported:

-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_list_implementations

```
[-vL] {-c <class context name> | -l <class LOID>}
[-debug] [-help]
```

Lists the implementation objects associated with the class named in <class LOID> or <class context path>. In its default setting the output will consist of each implementation object's LOID and architecture type.

The following options are supported:

-vL	Run the command in a verbose setting. The output will include byte size and a brief description of each object. Legion will use one extra method invocation per implementation object.
-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_list_instances

```
{[-c] <class context name> | -l <class LOID>}
[-debug] [-help]
```

Displays information about the instances of the class named by <context path> or <class LOID>. For every instance, the tool displays the class's LOID, current object address, status (active or inert), the host on which it resides, and the vault that holds its OPR. The example below shows that class `BasicFileClass` has two instances, and that both are currently running.

```
$ legion_list_instances /class/BasicFileClass
Class 1.01.66000000..000001fc0d63e97... knows about the
following instances:
LOID: 1.01.66000000.01000000.000001fc0a00...
Current oa: [xxx.xxx.xxx.xxx: 2020]
Current host: 1.01.07.30232908.000001fc0...
```

```

Current vault: 1.01.03.2e232908.000001fc0...
Status: object-running
LOID: 1.01.66000000.02000000.000001fc0edd...
Current oa: [xxx.xxx.xxx.xxx: 1895]
Current host: 1.01.07.31232908.000001fc0...
Current vault: 1.01.03.2e232908.000001fc0...
Status: object-running
$

```

The following options are supported:

-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_refresh_local_cache

Manually refresh some high-use objects' bindings and high-use contexts' LOIDs in your current login session.

legion_set_backup_vaults

```

{[-c] <instance context path> | -l <instance LOID>}
[-nodeac[tivate]]
{[-a | -d] [-c <vault1 context path> | -l <vault1 LOID>]
  [-a | -d] [-c <vault2 context path> | -l <vault2 LOID>] ...
  [-a | -d] [-c <vaultn context path> | -l <vaultn LOID>]}
| {-Cv <vault context path> -n <total number of vaults>}

```

Specify one or more backup vaults for an instance. Use `legion_skcc_set_class_vaults` to set backup vaults for a class (page 11).

Backup vaults are used to replicate an object's persistent state: the object's persistent state is replicated and stored in multiple vaults when it is deactivated, so that if its main vault crashes, its state is still available. However, the replication policy synchronizes the object's state when the object deactivates: if the main vault crashes while the object is still active, the backup vaults will not have the object's current state. If the object needs to be reactivated, it will use an out-of-date state.

Therefore, this command should only be used for objects that can tolerate potentially old states. This functionality is currently supported for the following classes: `ContextClass`, `BasicFileClass`, `AuthenticationObjectClass`, and `ImplementationObjectClass`.

You can either give a list of specific vaults or a context path from which a specified number of vaults will be selected. For example, to add `Vault2` and `Vault3` to object `Foo`'s list of backup vaults, you would enter:

```

$ legion_set_backup_vaults /home/mycontext/Foo \
  -a /vaults/Vault2 -a /vaults/Vault3

```

To add any two vaults from the /vaults context, you would enter:

```
$ legion_set_backup_vaults /home/mycontext/Foo \
-Cv /vaults -n 2
```

Use `legion_object_info` (page 10) to view an object's list of backup vaults.

The following options are supported:

-nodeac[tivate]	Do not deactivate the specified object when setting up its backup vaults
-a	Add the specified vault to the object's list of backup vaults
-d	Delete the specified vault
-Cv	Select the backup vaults from the specified context path
-n	Number of vaults to be selected

`legion_set_binding_agent`

```
[-unset] [-make_default] [-make_default_only]
{-l <object context path> | -c <object LOID>}
```

Set or unset the binding agent named in <object context path> or <object LOID> as the user's current and/or future login session binding agent.

The following options are supported:

-unset	Clear the current binding agent.
-make_default	Set the specified binding agent to be the user's binding agent for the current login session and any future login sessions.
-make_default_only	Set the specified binding agent to be the user's binding agent for the any future login sessions but do not set it as the current session's binding agent.

`legion_set_host`

```
{[-c] <instance context path> | -l <instance LOID>}
{[-c] <host context path> | -l <host LOID>} [-debug] [-help]
```

Calls the `set_host()` class-mandatory member function on the class of the instance named by in <instance LOID> or <instance context path>, causing the instance to migrate to the host named in <host LOID> or <host context path>.

In the example below, object Foo's host is changed from BootstrapHost to newHost.

```
$ legion_set_host -c Foo -c /hosts/newHost
```

The following options are supported:

- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

legion_set_vault

```
{[-c] <instance context path> | -l <instance LOID>}
{[-c] <vault context path> | -l <vault LOID>} [-permanent]
[-debug] [-help]
```

Migrates the state (OPR) of the object named in `<instance LOID>` or `<instance context path>` to the vault named in `<vault LOID>` or `<vault context path>`.

The following options are supported:

- permanent** Makes the migration permanent. I.e., the object's acceptable vault list will be change to list only the new vault. No further migration can occur without user intervention via `legion_instance_vault_list` (see page 49).
- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

Be careful with the `-permanent` flag. If none of the instance's acceptable hosts match the chosen vault (i.e., there are no compatible host-vault pairs) the instance cannot be reactivated. Alternatively, the chosen vault may be compatible only with hosts for which the class has no implementation.

Please note that this command can fail for many reasons, including:

- The instance does not exist.
- The user does not have permission to perform the migration.
- The target vault does not exist.
- The target vault will not grant the request to create a new OPR for the object.
- The source vault is unavailable or will not grant the request to get the current OPR.
- Other machine or system failure.
- The specified vault is unacceptable.

The class object will enforce the current restrictions on its instance's acceptable vaults before it migrates the object's state. You may need to change the instance's list of acceptable vaults (via the `legion_instance_vault_list` command) to include the target vault before running `legion_set_vault`.

This command is not exactly the equivalent of a true object migration, since the object is not reactivated after its state is moved. Where the object reactivates depends on the scheduling decision made at the time the reactivation occurs. If the `-permanent` flag is not used and the

instance's acceptable vault list is not otherwise altered, future activations of the instance may be on different vaults. If `-permanent` is used, all future activations must use the specified vault, until the acceptable vault list is altered.

`legion_synch_vaults`

```
{[-c] <instance context path> | -l <instance LOID>}  
[-nodeac[tivate]]
```

Synchronize the specified object's state in its main and backup vaults. The command copies the state contained in the object's main vault to all of its backup vaults. It will deactivate the object before synchronizing vault state unless the `-nodeactivate` flag is used.

If the object has been marked as Write-Once Read-Many (WORM) with the `legion_set_worm` command (page 12), its state does not change. You can still run this command for a WORM object, but it will not update its state.

This command is implicitly called upon object deactivation.

This functionality is currently supported for `ContextClass`, `BasicFileClass`, `AuthenticationObjectClass`, and `ImplementationObjectClass` objects.

The following option is supported:

<code>-nodeac[tivate]</code>	Do not deactivate the specified object when setting up its backup vaults.
-------------------------------------	---

2.3 Calls on LegionClass

`legion_add_class_mapping`

`<metaclass LOID> <class LOID> [-debug] [-help]`

This command notifies LegionClass that the meta-class named by `<metaclass LOID>` is the class of the class named by `<class LOID>`. LegionClass updates its class map accordingly.

The following options are supported:

- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

`legion_combine_domains`

`[-v] <list of domain cookie files> [-debug] [-help]`

Joins a set of Legion domains (systems) to create a single, larger, multi-domain Legion system. Before you run this command, you must obtain a copy of the Legion domain cookie files from all of the involved domains (see `legion_generate_domain_cookie`, page 24, and `legion_print_domain_cookie`, page 25), including the current Legion domain. If you are joining domain A to domain B and domain B has already been joined to domain C, you will need cookie files from domains A, B, and C. This provides transitivity in the system join. All three domains will be joined to one another to form a single system. The command is implemented this way to avoid communication failures; if a LOID can be passed to an object (e.g., in a method continuation list), that object should be able to use the LOID for further communication.

Non-transitive or non-reflexive joins would allow communication of LOIDs for which an object could not obtain a binding. For example, object X in domain A might be able to bind to object Y in domain B and pass a method to it, but object Y might not be able to bind to object X to pass it the return value.

In addition to joining the binding trees of the involved domains, `legion_combine_domains` also creates context links in the current domain's context space to all of the remote domains' root contexts. These links appear in local context space in the following path: `/domain/LegionDomain.<domain-id>`.

If the command is run on domains that are already connected, it has no affect and is harmless.

There is currently no mechanism supporting "unjoining" of domains. However, Legion security mechanisms (e.g., ACLs) can be used to effectively forbid any use of the current domain by outside domains.

The following options are supported:

- v** Provide a verbose output as the command is running.
- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

The example below combines two domains. If either had previously been connected to another domain, three cookie files would be listed.

```
$ legion_combine_domains LegionDomainCookie.35d82a07 \
    LegionDomainCookie.c8
Created 2 new domain interconnections
$
```

legion_create_implementation

```
<binary path name> <architecture>
{[-c] <class context name> | -l <class LOID>}
[-c <object context path>] [-nc] [-v] [-a <attribute>]
[-debug] [-help]
```

Creates an implementation object based on the binary executable named in <binary path name>. Each class maintains a list of the implementation objects that are suitable for its instances. There are a set of implementation objects created when your system was initialized (use `legion_ls -la /impl` to see a list). Several different implementation objects might be maintained by a class to support the use of multiple platforms – a class might have implementation objects for different architectures, for different operating systems, with different memory requirements, etc.

The new implementation object is associated with the class object named in <class LOID> or <class context path>, and is marked as usable for hosts of a specified type (linux, solaris, etc.). The sample below creates a new implementation object for my_class:

```
$ legion_create_implementation Legion/bin/linux/my_class \
    linux my_class
```

The new object is automatically assigned the context path `/impls/my_class.linux.1` (if you ran the example a second time, the new object would be called `/impls/my_class.linux.2`). You can use the `-c <object context path>` flag to specify a different context path or the `-nc` flag to specify that no context path be assigned.

At the moment, possible <architecture> values are:

- solaris (Sun Workstation/Solaris 5.7 or later)
- sgi_n32 (SGI Workstations/IRIX 6.5 n32 build or later n32 build)
- sgi_n64 (SGI Workstations/IRIX 6.5 n64 build or later n64 build)
- linux (x86/Red Hat 6.x Linux)
- alpha_linux (DEC Alpha/Red Hat Linux 6.x)
- alpha_DEC (DEC Alpha/OSF1 v4)
- rs6000 (IBM RS6000/AIX 4.3)
- hppa_hpux (HPUX 11.x)³

3 The HPUX 11 platform is available upon request. We will include an HPUX 10 platform in a future release.

- t90 (Cray T90/Unicos 10.x)⁴
- t3e (Cray T3E/Unicos/mk 2.x)

The following optional parameters are supported:

- c** <context path>
Specify a context path for the new object. Default is /impls/<binary_name>.<arch>.<#>
- nc**
Specify that the new object have no context name.
- v**
Run the command in verbose mode.
- a** <attribute>
Assign the new object an extra attribute.
- debug**
Catch and print Legion exceptions.
- help**
Print command syntax and exit.

legion_generate_domain_cookie

[-o <cookie output filename>] [-debug] [-help]

Generates a domain cookie file for the current Legion domain, as required by `legion_combine_domains` (page 22). A cookie file contains binding information for the LegionClass in the current domain, security credentials for joining to the current domain, and information about the context space of the current domain (for creating interdomain context space links). The default cookie file name is `LegionDomainCookie.<domain-id>`. In a secure environment you must be logged in as `/users/admin` for the current domain. This ensures that the required credentials can be generated and saved in the cookie file.

The following optional parameters are supported:

- debug**
Catch and print Legion exceptions.
- help**
Print command syntax and exit.

legion_init_arch

[-debug] [-help]

Creates and registers implementation objects for commonly used classes in the current architecture. This command is run on a new host to create its implementation objects in the proper place. Implementation objects for specific binary executables can be created with the `legion_create_implementation` utility (page 23).

The sample below, run on a linux machine, creates a set of implementation objects for a linux host.

```
$ legion_init_arch
Initializing Legion implementations for "linux"
Creating an implementation (ContextObject) for ContextClass
Continue (y=yes, Y=yes to all, n=no, N=no to all, v=verbose,
      V=verbose all)? Y
Creating an implementation (MetaClassObject) for LegionClass
Creating an implementation (ClassObject) for VanillaMetaClass
```

⁴ The T90 and T3E architectures must be used with virtual hosts. Please see section 14.0 in the System Administrator Manual.

```

Creating an implementation (BindingAgent) for
    BindingAgentClass
Creating an implementation (BasicFileObject) for
    BasicFileClass
Creating an implementation (ttyObject) for ttyObjectClass
Creating an implementation (StatTreeObject) for StatTreeClass
$

```

The following options are supported:

-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_list_domains [-debug] [-help]

List the domains currently connected to your current Legion domain. The output will list the binding for your current domain and any domains linked to your current domain. Please see `legion_combine_domains` (page 22) for information about connecting Legion domains.

The following options are supported:

-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_print_domain_cookie
 [-i <cookie input file>] [-debug] [-help]

Displays the contents of a Legion domain cookie file (required for using `legion_combine_domains`). By default the command displays the contents of the file `LegionDomainCookie.<current-domain-id>`, but any input file name can be specified using the `-i` option.

The following options are supported:

-i <cookie input file>	Specify the path of the cookie file to print.
-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

2.4 Calls on file and context objects

legion_activate_instances

```
{[-c] <class context name> | -l <class LOID>} [-debug] [-help]
```

Activates all instances of the class named in <class context path> or <class LOID>. Instances that are already active will be unaffected.

The following options are supported:

-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_allow_activation

```
[-entire_class] [-debug] [-help]  
{[-c] <context path> | -l <LOID>}
```

Activates the object or instances of the class named in <context path> or <LOID>. It should be used in association with the `stay_down` option of `legion_deactivate_object` or `legion_deactivate_instances` (page 16 and page 29, respectively).

The following options are supported:

-entire_class	Unlock all of the class's instances. You must be the class's owner in order to use this option.
-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_cat

```
<context path1> <context path2> ... <context pathN>  
[-debug] [-help]
```

Prints the contents of the Legion file object(s) named in <context path> to standard output. It is similar to the Unix `cat` command.

```
$ legion_cat newFileObject  
blah, blah, blah.  
$
```

The following options are supported:

-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_cd

```
<context path> [-debug] [-help]
```

Changes the current working context to the context named in <context path>. Note that the path name can be relative or absolute. This command is analogous to the Unix `cd` command.

The following options are supported:

-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_context_add

<object LOID> <context name> [-debug] [-help]

Adds a new name <context name> for the object named in <object LOID> to the current context space.

The following options are supported:

-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_context_create

<context path> [-debug] [-help]

Creates a new context in Legion context space, and assigns it the name given in <context path>. This command is analogous to the Unix `mkdir` command. The new context will be placed in your current context.

The following options are supported:

-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_cp

**[-r] [-v] [-m] [-p] [-localsrc] [-localdest]
[-V <vault context path>] <source path> <destination path>
[-debug] [-help]**

Copies the file object or tree named in <source path> (named as either a context path or a local path) to a new, duplicate, file object or tree named in <destination path> (named as either a context path or a local path). Use the recursive mode (-r) to export objects to local file space. Analogous to the Unix `cp` command. For example, to copy the file object `Foo` to `/home/myContext/Bar`, you would enter:

```
$ legion_cp Foo /home/myContext/Bar
```

This copies the contents of `Foo` in the new file object `Bar` in the `/home/myContext` path.

You can now use wildcards in the <source path> parameter (for context space or local file space). Please note that you must escape the `"*"` character. The example below copies all files that start with `"Foo"` from local directory space into the `/home/myContext` context.

```
$ legion_cp -localsrc Foo\* /home/myContext
```

The new file objects are assigned the same names (e.g., if `Foo\*` finds `Foo1` and `Foo2`, the duplicate file objects will be at `/home/myContext/Foo1` and `/home/myContext/Foo2`).

Note that you cannot use wildcards in a destination path and you cannot copy multiple files to a single file object.

The following optional parameters are supported:

- v** Verbose mode. Prints information about which files and directories the command is currently working on.
- r** Recursive mode. If the <source path> is a directory, all of its contents are copied recursively. Only files and contexts/directories are handled. If other objects are encountered, they are skipped and `legion_cp` prints a warning message. Note that recursive mode automatically detects cycles in context space and prevents the recursive copy from revisiting context nodes in the cycle. A warning message is printed in the event that cycles are detected.
- localsrc** Local source mode. Use to indicate that the file or directory to be copied (<source path>) is in the local file system. By default the <source path> is assumed to be in Legion context space.
- localdest** Local destination mode. This option indicates that the <destination path> is in the local file system. The path is assumed to be a Legion context path.
- V <vault context path>** Specify a vault restriction for new objects created by this command. Supply the context path of the vault that should manage new objects created as `legion_cp` runs.
- m** Match-class mode. This mode indicates that when files or contexts are created by this command, they should match the class of their source context or file. By default, new files and contexts are created using the default file and context classes for your current Legion environment. This mode can only be used when copying within Legion context space (i.e., without `-localsrc` or `-localdest`).
- p** Print out the size, transfer time, and transfer rate for each file copied.

-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_deactivate_instances

```
[-stay_down] [-debug] [-help]
{[-c] <class context name> | -l <class LOID>}
```

Deactivates all instances of the class named in <class context path> or <class LOID>. Instances that are already deactivated will be unaffected.

The following options are supported:

-stay_down	Forces instances to stay inactive. They can only be reactivated by <code>legion_allow_activation</code> (see page 26).
-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_destroy_instances

```
{[-c] <class context name> | -l <class LOID>} [-debug] [-help]
```

Destroys all instances of the class named in <class context path> or <class LOID>. Any active instances are deactivated.

This command will remove the LOIDs of the specified class' instances in all contexts, not just the current context. However, it will not remove any context name associated with this object: you must use the `legion_rm` command to remove the object's name(s), or you will get binding errors. (You can use `legion_ls -A` to check for multiple context names.)

The following options are supported:

-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_direct_output

```
{[-c] <object context path> | -l <object LOID>}
{[-c] <tty context path> | -l <tty LOID>} [-debug] [-help]
```

Causes the object named in <object path> or <object LOID> to direct its output (standard out and standard error) to the Legion tty object named by <tty context path> or <tty LOID>. Note that this command can only be invoked on objects that have dynamic output redirection enabled. If the command is invoked on an object that does not have redirection enabled, neither the object nor the tty is affected and an error message is displayed.

The following options are supported:

-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_export_dir

[-v] [-help] [-noserver] [-autorehash]
<local directory path> <target context path>

The **legion_export_dir** command allows a complete directory tree in the local file system to be temporarily linked into Legion context space. A new context is created to replicate the local directory tree in context space. The files contained in the shared directory are copied in to Legion objects in the new context. The new context objects have the same names as the original files. For example, if the local directory contains three files called **foo.txt**, **foo.doc**, and **foo.c**, the new context will contain three Legion objects called **foo.txt**, **foo.doc**, and **foo.c**.

The command executes for as long as you wish: if you pause the command the tree's context space will not be available. Once you resume the command, the context will again be available.

To pause the command, run **legion_export_dir_quit** (below) or (if the command was started in **-noserver** mode) press **^C**. To resume using the exported files, re-execute the command, specifying the same local base directory path and target context path. You can also run **legion_rm -r** on the new context, which will recursively delete the context and its contents, to stop the command.

You can make changes to the shared local directory while the command is running, but those changes won't be reflected in context space unless you use the **-autorehash** flag or run **legion_export_dir_rehash** (below).

The following parameters are required:

<local directory path> A local directory or directory tree that you wish to export to Legion context space.

<target context path> A new context path which will hold copies of the exported directory tree.

The following options are supported:

-v Verbose mode. Prints information about which files and directories the command is currently exporting.

-help Print command syntax and exit.

-noserver Keep the program attached to a terminal so that it can be shutdown via **^C** and its output can be monitored.

-autorehash Periodically check the shared local directory so that any changes are reflected in context space.

`legion_export_dir_quit`
`<target context path>`

To be used in conjunction with `legion_export_dir` (above). Pauses `legion_export_dir` for the specified context path.

The following parameter is required:

`<target context path>` A context path holding copies of an exported directory tree.

`legion_export_dir_rehash`
`<target context path>`

To be used in conjunction with `legion_export_dir` (page 30). Tells the command to check the shared local directory for any changes and to update the shared context space accordingly.

The following parameter is required:

`<target context path>` A context path holding copies of an exported directory tree.

`legion_get_host`
`{[-c] <object context name> | -l <object LOID>} [-debug] [-help]`

This command looks up and returns the LOID of the host on which the object named in `<object context path>` or `<object LOID>` currently resides.

```
$ legion_get_host Foo
1.01.07.d49d1a40.000001fc0c04724...
$
```

The following options are supported:

-debug Catch and print Legion exceptions.
-help Print command syntax and exit.

`legion_get_vault`
`{[-c] <object context name> | -l <object LOID>} [-debug] [-help]`

Returns the LOID of the vault which the object named in `<object context path>` or `<object LOID>` is currently using to store its OPR.

The following options are supported:

-debug Catch and print Legion exceptions.
-help Print command syntax and exit.

`legion_list_names`
`{[-c] <object context name> | -l <object LOID>} [-debug] [-help]`

Lists all of the given object's context names. This includes names assigned by other users. The following options are supported:

-debug Catch and print Legion exceptions.
-help Print command syntax and exit.

legion_ln

<context path> <new alias> [-debug] [-help]

Assigns an additional name, given in **<new alias>**, to the object named in **<context path>**. Analogous to the Unix **ln** command. Path names can be relative or absolute. An object can have multiple context names, assigned by one or more users. The same context name can be assigned to different objects or to the same object so long as the contexts names are in different contexts (just as the same file names can be used in different levels of a Unix directory).

The following options are supported:

-debug Catch and print Legion exceptions.
-help Print command syntax and exit.

legion_ls

[-laLRAdqv] [<context path>] [-debug] [-help]

Lists the contents of a named Legion context in ascii-alphabetical order. Note that the pathname can be relative or absolute. The command is analogous to the Unix **ls** command. The default setting lists the current context. You can include a context path in the **<context path>** parameter to list the contents of that context. For example:

```
$ legion_ls /hosts
BootstrapHost
my.host.DNS.name
$
```

You can get more specific information (object type, LOID, other context aliases, state) about objects with the flags. For example, you can get more information about a particular context's objects with the **-l**, **-a**, and **-A** flags. The output below shows a complete list of all objects listed in this context, the objects' types, and any context aliases associated with each listed object.

```
$ legion_ls -laA /hosts
.                               (context)
  /hosts
..                              (context)
  /class/..
  /hosts/..
  /vaults/..
  /home/..
BootstrapHost                  (object)
  /hosts/BootstrapHost
  /hosts/host.DNS.name
host.DNS.name                  (object)
  /hosts/BootstrapHost
  /hosts/host.DNS.name
$
```

According to this, there are four names listed in `/host s`, two referring to contexts and two to objects. We can see from the alternative context names, though, that `BootstrapHost` and `my . host . DNS . name` refer to the same object.

You can now use wildcards in the `<context path>` parameter. Please note that you must escape the `"*"` character. I.e., to search for context objects containing "foo" in their names you would enter:

```
$ legion_ls \*foo\*
```

You cannot use wildcards with the `-R` flag.

Optional parameters do the following:

- l** List object type and information, if available. Objects of unknown type will be listed as `object` and faulty objects will be listed as `not available`.
- a** List "hidden" objects in the context, i.e. those objects whose names begin with a `"."` character. Examples would be the `"."` (current) and `".."` (parent) contexts.
- L** List the LOID associated with each entry.
- R** Print a recursive list of the current context. You cannot use this flag with wildcards.
- A** List all known context aliases for each listed object.
- d** List contexts like other objects, rather than listing their contents.
- q** When creating a long list, do not activate inactive objects.
- v** Run command in verbose mode.
- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

```
legion_mkdir
  <context path>
  [-debug] [-help]
```

Creates a new context in Legion context space, and assigns it the name given in `<context path>`. This command is analogous to the Unix `mkdir` command (it is also identical to the `legion_context_create` command). The new context will be placed in your current context. The output will contain the new context's location and its LOID.

The following options are supported:

- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

legion_mv

<context path> <new context path>
 [-debug] [-help]

Assigns a new context name, given in <new context path> to the object named in <context path>. Pathnames can be relative or absolute. Analogous to the Unix mv command.

The following options are supported:

-debug Catch and print Legion exceptions.
-help Print command syntax and exit.

legion_pwd

[-debug] [-help]

Prints your current context path. Similar to the Unix pwd command.

The following options are supported:

-debug Catch and print Legion exceptions.
-help Print command syntax and exit.

legion_rm

[-r] [-f] [-v] <context path list>
 [-debug] [-help]

Removes the context path[s] named in <context path list> from Legion context space. Pathnames can be relative or absolute. Analogous to the Unix rm command.

If the context path listed is the last (i.e., only) name mapped to a given object, the object will be destroyed.

You can now use wildcards in the <context path list> parameter. Please note that you must escape the "*" character. I.e., to remove all context objects containing "foo" in their names you would enter:

```
$ legion_rm \*foo\*
```

Optional parameters do the following:

-r Recursively remove one or more contexts and all of their contents.
-f Forcibly remove faulty objects (those with bad bindings).
-v Run this command in a verbose setting. This is useful for checking whether or not an object is removed.
-debug Catch and print Legion exceptions.
-help Print command syntax and exit.

legion_set_tty

<tty context path> [-debug] [-help]

This command will set an environment variable to indicate which tty object should be used by subsequent programs. (Please see page 75 in the Basic User Manual for a discussion of tty objects.) By selecting a new current tty object, users can redirect the output to any window or file.

```
$ legion_set_tty /context_path/my-tty
```

Note that program output does not have to be directed to the same window in which the program is run. By setting a new current tty object, the output can be redirected to any window, or even a file. For example:

```
$ legion_create_object -c /class/ttyObjectClass my-tty
$ legion_set_tty /log-file
```

creates a tty object whose output is sent to a file. To view the tty output, use the `legion_tty_watch` command (below).

The following options are supported:

-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_tty

<tty context path>
[-debug] [-help]

This command can be used to direct all output from a particular shell back to that shell (i.e., create and set a tty object for a shell) in one step, rather than running `legion_create_object`, `legion_set_tty`, and `legion_tty_watch`. If no tty object exists at the path named in <tty context path> it creates a new object, sets it as the target tty, and starts the `legion_tty_watch` process in the background of the shell in which the command was run. If a tty object already exists at the named context path, the command sets that tty object as the target. Please see page 75 in the Basic User Manual for a discussion of tty objects.

The following options are supported:

-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_tty_off

[-debug] [-help]

Unsets (stops) the Legion tty object for the current shell so that Legion programs executed in that shell after `legion_tty_off` is run will not display their output to a Legion tty object. This command also shuts off the background `legion_tty_watch` process for the current shell.

This command only works on tty objects that have been set with the `legion_tty` command. See `legion_tty_watch` for information on turning off other tty objects. Please see page 75 in the Basic User Manual for a discussion of tty objects. Note that this does not destroy the tty object. The object can be reused with `legion_tty`, `legion_set_tty`, or `legion_tty_watch`.

The following options are supported:

- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

```
legion_tty_redirect
  <object context path>
  [-debug] [-help]
```

Causes the Legion tty object currently set in the shell environment to stream directly into the file object named in `<object context path>`. If the file does not already exist the system creates it. Existing files are appended to, not truncated. A single tty object can be simultaneously directed into any number of files (as well as watched from any number of terminal windows). Please see page 75 in the Basic User Manual for a discussion of tty objects.

The following options are supported:

- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

```
legion_tty_unredirect
  <object context path>
  [-debug] [-help]
```

Causes the Legion tty object currently set in the shell environment to stop streaming into the file object named in `<object context path>`. If the tty object is not currently directing output to the named file the command is ignored. Please see page 75 in the Basic User Manual for a discussion of tty objects.

The following options are supported:

- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

```
legion_tty_watch
  [[-c] <tty context path> | -l <tty LOID>]
  [-debug] [-help]
```

Causes output written to a Legion tty object to be printed to standard output. If no command line parameters are specified, the current tty object set for the shell session is selected. Otherwise, the tty object named in `<tty LOID>` or `<tty context path>` is selected. Note, the command will not self-terminate: to stop the program send it a SIGINT (i.e., using `^C` or "kill -INT"). Any number of `legion_tty_watch` sessions may simultaneously watch the same Legion tty object. Please see page 75 in the Basic User Manual for a discussion of tty objects.

The following options are supported:

- | | |
|---------------|------------------------------------|
| -debug | Catch and print Legion exceptions. |
| -help | Print command syntax and exit. |

`legion_2drm`
 <file name>

This command removes a 2D file object from Legion. It differs from `legion_rm` in that the object's subfiles are removed as well. If you use `legion_rm` to remove a 2D file object, the object's subfiles will not be removed.

The following parameter must be included:

<code>file name</code>	Local path of a 2D file object.
------------------------	---------------------------------

2.5 Start-up and shutdown functions

legion_add_host_account

```
{-l <host object LOID> | [-c] <host object context path>}
{[-f <mapping file name>] |
  [<Unix user id> [-l <owner LOID> | -c <owner context path>]]}
[-debug] [-help]
```

This command is used on PCD host objects. Adds a mapping between a Legion account and a Unix account and adds this mapping to a PCD host object's list of available accounts. The user's Unix user id is named in the <Unix user id> parameter. The host object is named in the <host object LOID>/<host object context path> parameter. The user's Legion user id can be given in the <owner LOID>/<owner context path> parameter or listed in the mapping file.

The following options are supported:

- f <mapping file name>** Names a mapping file. The mapping file contains a list of Unix user id-Legion user name mappings for that PCD host object. There is no limit on the number of mappings that can be listed. One mapping per line. If a Unix id is listed but not mapped to a Legion user id, the account will be a guest account.
- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

There are two ways to add host accounts: directly from the command line or, via the -f flag, with a mapping file. A mapping file contains a list of Unix-Legion account mappings. This file contains a list of Unix user ids and any corresponding Legion user ids. If no Legion account is named, the account will be treated as a guest account. There is no limit on the number of mappings that can be listed.

A sample mapping file is below.

guest	
unixLucy	-c /users/lucy
unixJohn	-c /users/john

legion_create_class

```
[[[-c] <context path>] [-sc <scheduler context path>]
[-sl <scheduler LOID>]
[-debug] [-help]
```

Creates a new instance of VanillaMetaClass. The new class will be placed in the current working context unless you specify otherwise.

The following options are supported:

-c <context path>	Assigns the context name in <context path> to the new instance.
-sc <scheduler context path>	Specify the context path of the default scheduler object that the new class should use.
-sl <scheduler LOID>	Specify the LOID of the default scheduler object that the new class should use.
-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_destroy_host

[-v] {[-c] <host context path> | -l <host LOID>} [-debug] [-help]

Destroy a given host object. All contexts objects on that host will be destroyed and all active objects will be deactivated. Legion will automatically search your context space and remove any dangling context names for the host object.

Optional parameters do the following:

-v	Run in verbose mode.
-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_destroy_vault

{[-c] <vault context path> | -l <vault LOID>} [-debug] [-help]

Destroy a given vault object. Legion will attempt to move all of the vault's current OPRs off of the vault object and then destroy the vault object. If any OPRs cannot be successfully moved the process will abort and an error message will be displayed.

The following options are supported:

-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_initialize

Populates the Legion system with basic classes and implementations. This command should be run after a Legion system is started for the first time (using `legion_startup`). On subsequent activations of the system, the state created by this utility will already exist, so this command should not be run again.

legion_list_host_accounts

```
{-l <host object LOID> | [-c] <host object context path>}
<user id> [-debug] [-help]
```

This command is used on PCD host objects. It lists the available accounts on a host object. If no host object argument is provided, your current host object will be used as a default.

The following options are supported:

- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

legion_make_setup_script

```
[-o <script basename>] [-OPR <OPR dir name>]
[-L <$LEGION dir name>] [-debug] [-help]
```

Generates a Legion setup script for your system. This script sets the environment variables for Legion users.

The following options are supported:

- o <script basename>** Specify the basename for the resulting setup scripts (default is /home/xxxx/OPR/setup). This command will generate two setup scripts, for /bin/sh derivative users and csh-derivative users. The scripts will be named <script basename>.sh and <script basename>.csh, respectively.
- OPR <OPR dir name>** Specify the OPR directory name that will be set up when the resulting scripts are run. This directory will contain the user's local copy of Legion-Class.config (default is "Legion-OPR"). The user's local version of the directory will be placed in the user's \$HOME.
- L <\$LEGION dir name>** Specify the value of \$LEGION, which is the directory where the resulting scripts are run. The default is the current value of \$LEGION.
- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

legion_print_config

```
[-debug] [-help]
```

Prints the "well-known" binding for LegionClass in the current Legion configuration. The following options are supported:

- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

legion_remove_host_account

[-l <host object LOID> | [-c] <host object context path>]
[-debug] [-help]

This command is used on PCD host objects. It removes one or more account mappings from the host object's list of available accounts. The <user id> parameter is the user's Unix user id.

If no host is named in the <host object LOID>/<host object context path> parameter, your current host object will be the default host.

The following options are supported:

-debug Catch and print Legion exceptions.
-help Print command syntax and exit.

legion_setup_state

[-i] [-debug] [-help]

Creates OPRs for the basic Legion system objects. This script should be run when starting a Legion system for the first time. The following optional parameters are supported:

-i Run the command in an interactive mode.
-debug Catch and print Legion exceptions.
-help Print command syntax and exit.

legion_shutdown

[-local] [-f] [-i]
[-debug] [-help]

Shuts down a running Legion system, preserving the state of all objects for subsequent reactivation of the system. Optional parameters allow users to shut down individual hosts and to specify an interactive shutdown.

Optional parameters do the following:

-local Shut down only a local host or vault.
-f Force the termination of a system. This may leave processes running, however, and prevent a system restart.
-i Put the shutdown in an interactive mode, which provides prompts for user actions.
-debug Catch and print Legion exceptions.
-help Print command syntax and exit.

legion_shutdown_class

{[-c] <class context name> | -l <class LOID>}
[-debug] [-help]

Deactivates the class object named in <class LOID> or <context path> and all of its instances. This command operates recursively: if applied to a metaclass, for example, it would deactivate the metaclass, all of its class instances, all of their instances, etc.

The following options are supported:

- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

legion_starthost

```
[-L <$LEGION>] [-O <$LEGION_OPR>] [-A <$LEGION_ARCH>]
[-B <path>] [-N <context name>] [-U <user id>] [-C <host class>]
<new host name> [<compatible vault list>]
[-debug] [-help]
```

Creates a new host object on the specified <new host name>, using the `legion_create_object_r` command (page 15, automatically invoked on the host class). The <new host name> is the host's DNS name. The `legion_starthost` command selects the following default values for the new object:

<\$LEGION_OPA> = `$LEGION_OPR/Host-$HOST.OPA`

<binary path> = `$LEGION/bin/$LEGION_ARCH/UnixHostObject`



This command uses remote shell (`rsh` or `ssh`) classes to start a new host object on a specified host. Please note that you **must** be able to run `rsh/ssh` on the target host from your current machine without having to enter a password. You can set up an `.rhosts` file for `rsh` or an `authorized_keys` file for `ssh` to accomplish this: see the `rsh` and `ssh` man pages for further information.

You can run Legion commands on a remote host using `rsh` or `ssh`, once you set the proper environment variables. For `sh`, `ksh`, or `bash`, use:

```
LEGION_RSH=<rsh|ssh>
LEGION_RCP=<rcp|scp>
export LEGION_RSH LEGION_RCP
```

For `csh`, use:

```
setenv LEGION_RSH <rsh|ssh>
setenv LEGION_RCP <rcp|scp>
```

Optional parameters do the following:

- L <\$LEGION>** Specify `$LEGION` for host (default is local `$LEGION` value).
- O <\$LEGION_OPR>** Specify `$LEGION_OPR` for host (default is local `$LEGION_OPR` value).
- A <\$LEGION_ARCH>** Specify the architecture type for the host (default is local `$LEGION_ARCH` value).
- B <path>** Specifies the basename of the executable host program that will be started on the target host (default is "UnixHostObject").

- N** <context name> Specify the context name for the host object (default is /hosts/<host name>).
- U** <user id> Specify the user id for host (default is current user id).
- C** <host class> Specify the context path of the host class to use (default is /class/UnixHostClass).
- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

legion_startup

[-local] [-debug] [-help]

Starts up basic Legion services. The following optional parameters are supported:

- local** Starts up only a local host or vault.
- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

legion_startvault

**[-L <\$LEGION>] [-O <\$LEGION_OPR>] [-A <\$LEGION_ARCH>]
 [-N <context name>] [-U <user id>] [-C <vault class>]
 <host name> [<compatible host list>]
 [-debug] [-help]**



Creates a new vault object on the specified <host name>, using the `legion_create_object_r` command (automatically invoked on the host class) (page 15). The <host name> is the host's DNS name.

This command uses remote shell (`rsh` or `ssh`) classes to start a new vault object on a specified host. Please note that you *must* be able to run `rsh/ssh` on the target host from your current machine without having to enter a password. You can set up an `.rhosts` file for `rsh` or an `authorized_keys` file for `ssh` to accomplish this: see the `rsh` and `ssh` man pages for further information.

You can run Legion commands on a remote host using `rsh` or `ssh`, once you set the proper environment variables. For `sh`, `ksh`, or `bash`, use:

```
LEGION_RSH=<rsh|ssh>
LEGION_RCP=<rcp|scp>
export LEGION_RSH LEGION_RCP
```

For `csh`, use:

```
setenv LEGION_RSH <rsh|ssh>
setenv LEGION_RCP <rcp|scp>
```

The following optional parameters are supported:

- L** <\$LEGION> Specify \$LEGION for the vaults host. Default is the local \$LEGION value.
- O** <\$LEGION_OPR> Specify \$LEGION_OPR for the vaults host. Default is the local \$LEGION_OPR value..
- A** <\$LEGION_ARCH> Specify the architecture of the vaults host. Default is the local \$LEGION_ARCH value.
- N** <context name> Specify the context name for the vault object. Default is /vaults/vault-<host name>.
- U** <user id> Specify the user id to use on the vaults host. Default is current user id.
- C** <vault class> Specify the vault object's context path (default is /class/UnixVaultClass)
- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

2.6 Scheduling support

legion_add_sub_collection

```
{-l <collection LOID> | -c <collection path>}
{-l <member LOID> | -c <member path>} [-q <query>]
```

This command creates a new parent-subcollection relationship. A subcollection is a collection object that is polled by another collection object (called the parent collection) with a specified MESSIAHS-type query (see `legion_query_collection`, page 51, for details). If no query is specified, the default value of 'true' will be used. You can run this command multiple times to start multiple queries on a single subcollection. A parent collection can have more than one subcollection and can run multiple queries on a single subcollection. A subcollection can have more than one parent as well as its own subcollections.

You must designate a parent collection (named in `<collection LOID>` or `<collection path>`). The member (named in `<member LOID>` or `<member path>`) is the subcollection. Both collections must already exist.

You can edit the `collection_update_frequency_secs` attribute with the `legion_update_attributes` command (page 12). The default setting is 300 seconds. The example below will set the default collection to update itself every 10 minutes.

```
$ legion_update_attributes -c /etc/Collection \
"collection_update_frequency_secs(600)"
```

The following option is supported:

-q <query> A MESSIAHS-style query that will be run on the subcollection. The default selection is 'true'. See `legion_query_collection` (page 51) for details.

legion_class_host_list

```
[[ -c ] <class context name> | -l <class LOID>]
[ { -a | -d | -t } <host1> <host2> ... <hostn> ]
[ -p ]
[ -debug ] [ -help ]
```

Manipulates the list of hosts upon which the class named in `<class context name>` or `<class LOID>` can place its instances. The list of acceptable hosts for a given class consists only of hosts that have been added to the list. The list therefore may not necessarily include all possible hosts. If there are no hosts listed as acceptable the user can assume that all hosts are acceptable.

Note also that if you identify the class by its context name in the first parameter (i.e., `myClass`) you must use context names in the `<host1> <host2> ... <hostn>` parameter. If you use a LOID (with the `-l` flag) you must use LOIDs.

The example below adds a new host to the list of acceptable hosts of BasicFileClass, using the `-A` flag.

```
$ legion_class_host_list /class/BasicFileClass \
-a /hosts/newHost
** ADDED 1 host(s) to class's acceptable host set
```

The `-p` flag can then be used to check the listing.

```
$ legion_class_host_list /class/BasicFileClass -p
** ACCEPTIBLE HOST LISTING:
** 1.01.07.d59d1a40.000001fc094e23...
$
```

The following optional parameters are supported:

- l** Use dotted hex LOIDs to specify class and host.
- a** Add named host to the class's acceptable host list.
- d** Delete named host from the class's acceptable host list.
- t** Test whether or not a host is on the class's acceptable host list.
- p** Display the class's acceptable host list.
- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

You can use the `-a`, `-d`, and `-t` flags more than once when running the command but regardless of how you list them on the command line Legion will process them in a specific order when you run the command: first adding any new hosts, then deleting old hosts, then testing any hosts, and finally printing out the results.

legion_class_vault_list

```
[[-c] <class context path> | -l <class LOID>]
[{-a | -d | -t} <vault1> <vault2> ... <vaultn>] [-p]
[-debug] [-help]
```

Manipulates the list of vaults upon which the class named in `<class context path>` or `<class LOID>` can place its instances' OPRs.

Note that if you identify the class by its context name in the first parameter (i.e., `myClass`) you must use context names in the `<vault1> <vault2> ... <vaultn>` parameter. Similarly, if you use a LOID (with the `-l` flag) you must use LOIDs.

Optional parameters are:

- l** Use LOIDs to specify class and vault.
- a** Add named vault to the class's acceptable vault list.
- d** Delete named vault from the class's acceptable vault list.
- t** Test whether or not a vault is on the class's acceptable vault list.

- p** Display the class's acceptable vault list.
- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

You can use the **-a**, **-d**, and **-t** flags more than once when running the command but regardless of how you list them on the command line Legion will process them in a specific order when you run the command: first adding any new vaults, then deleting old vaults, then testing any vaults, and finally printing out the results.

legion_config_scheduler

```
{[-c] <scheduler context path> | -l <scheduler LOID>}
[-get_enactor] [-get_collection]
[-set_enactor {[-c] <enactor context path> | -l <enactor LOID>}]
[-set_collection {[-c] <collection path> | -l <collection LOID>}]
[-debug] [-help]
```

This command can be used to query or configure the helper objects used by a basic Legion scheduler. Basic Legion scheduler objects use a collection helper object to obtain information about available resources upon which they can schedule objects. After constructing a schedule, basic Legion schedulers use an enactor helper object to implement scheduling decisions (to actually start up the scheduled objects). Use this command to assign a particular collection and enactor to a basic Legion scheduler or vice versa.

The following optional parameters are supported:

- get_enactor** Print the LOID of a basic Legion scheduler's currently assigned enactor helper object.
- get_collection** Print the LOID of a basic Legion scheduler's currently assigned collection helper object.
- set_enactor** Set the enactor named in <enactor LOID> or <enactor path> to the scheduler.
- set_collection** Set the collection named in <collection LOID> or <collection path> to the scheduler.
- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

legion_host_vault_list

```
{[-c] <host context path> | -l <host LOID>}
[{ -a | -d | -t } <vault1> <vault2> ... <vaultn>] [-p]
[-debug] [-help]
```

Used to display and manipulate list of vaults with which the host named in <host context path> or <host LOID> can interoperate. Note also that if you identify the host by its context name in the first parameter (i.e., as *foo*) you must also identify the vaults by their context names in the <vault1> <vault2> ... <vaultn> parameter. Conversely, if you use a LOID (via **-l**) you must identify the vaults by their LOIDs.

Use the `-p` flag to list the vaults that a host can operate on:

```
$ legion_host_vault_list /hosts/HostName -p
** COMPATIBLE VAULT LISTING:
** 1.01.03.d49d1a40.000001fc0a69cbb845...
$
```

The following optional parameters are supported:

- l** Use dotted hex LOIDs to specify host and vault.
- a** Add named vault to the host's acceptable vault list.
- d** Delete named vault from the host's acceptable vault list.
- t** Test whether or not a vault is on the host's acceptable vault list.
- p** Display the host's acceptable vault list.
- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

You can use the `-a`, `-d`, and `-t` flags more than once when running the command but regardless of how you list them on the command line Legion will process them in a specific order when you run the command: first adding any new vaults, then deleting old vaults, then testing any vaults, and finally printing out the results.

`legion_instance_host_list`

```
{[-c] <instance context name> | -l <instance LOID>}
[{-a | -d | -t} <host1> <host2> ... <hostn>] [-p]
[-debug] [-help]
```

Manipulates the list of acceptable hosts upon which the instance can be placed. Note that if you use identify the instance by its context path you must use context paths in the `<host1> <host2> ... <hostn>` parameter. Conversely, if you identify the instance by its LOID (with the `-l` flag) you must use LOIDs in the `<host1> <host2> ... <hostn>` parameter.

The following optional parameters are supported.

- l** Use LOIDs to specify instance and host.
- a** Add named host to the instance's acceptable host list.
- d** Delete named host from the instance's acceptable host list.
- t** Test whether or not a host is on the instance's acceptable host list.
- p** Display the instance's acceptable host list.
- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

You can use the `-a`, `-d`, and `-t` flags more than once when running the command but regardless of how you list them on the command line

Legion will process them in a specific order when you run the command: first adding any new hosts, then deleting old hosts, then testing any hosts, and finally printing out the results.

```
legion_instance_vault_list
{[-c] <instance context name> | -l <instance LOID>}
[{-a | -d | -t} <vault1> <vault2> ... <vaultn>] [-p]
[-debug] [-help]
```

Manipulates the list of acceptable vaults upon which the instance can be placed. Note that if you identify the instance by its context path you must use context paths in the <vault1> <vault2> ... <vaultn> parameter. If you use a LOID (with the -l flag), you must use LOIDs in the <vault1> <vault2> ... <vaultn> parameter.

The following optional parameters are supported.

-l	Use LOIDs to specify instance and vault.
-a	Add named vault to the instance's acceptable vault list.
-d	Delete named vault from the instance's acceptable vault list.
-t	Test whether or not a vault is on the instance's acceptable vault list.
-p	Display the instance's acceptable vault list.
-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

You can use the -a, -d, and -t flags more than once when running the command but regardless of how you list them on the command line Legion will process them in a specific order when you run the command: first adding any new vaults, then deleting old vaults, then testing any vaults, and finally printing out the results.

```
legion_join_collection
{[-c] <collection context path> | -l <collection LOID>}
{[-c] <member context path> | -l <member LOID>}
[-debug] [-help]
```

This command joins the objects named in <member LOID> or <member path> to the collection named in <collection LOID> or <collection path>. Any Legion object can be added to a collection.

You can use the `legion_query_collection` command (page 51) to get information (in the form of object attributes) about the given collection.

The following options are supported:

-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_leave_collection

```
{[-c] <collection context path> | -l <collection LOID>}
{[-c] <member context path> | -l <member LOID>}
[-debug] [-help]
```

This command removes the object named in <member LOID> or <member path> from the collection named in <collection LOID> or <collection path>.

The following options are supported:

-debug Catch and print Legion exceptions.
-help Print command syntax and exit.

legion_list_oprs

```
{[-c] <vault context path> | -l <vault LOID>}
[-debug] [-help]
```

Lists all OPRs currently stored in a given vault. The output includes information about inert objects' LOIDs, OPA, owner, and status.

The following options are supported:

-debug Catch and print Legion exceptions.
-help Print command syntax and exit.

legion_list_sub_collections

```
{-l <collection LOID> | -c <collection path>}
```

List a collection's set of subcollections and queries.

legion_make_schedule

```
{-c <class context path> | -l <class LOID>}
[-n <number of nodes>] [-f <specification file>]
[-q <query>] [-help]
```

This command creates a host file for running an instance of the class named in <class context path> or <class LOID>. The host file is printed to stdout. The output of `legion_make_schedule` may be redirected into a file and used with the `-HF` flag in `legion_mpi_run` (see page 85).

The following options are supported:

-n <number of nodes>
 Specify the number of nodes required for this run.
 Default is 1.

-f <specification file>
 Name a specification file that contains relative performance ratios for different architectures. E.g., the specification file could contain the lines:

```
linux 1
alpha_linux 1.5
solaris 0.7
```

```
-q <query>    Specify a MESSIAHS query (interface described
               below in legion_query_collection) that can
               be used to prune the list of hosts on which the run
               can be scheduled.

-h help        Print command syntax and exit.
```

legion_query_collection

```
[-v]
{[-c] <Collection context path> | -l <Collection LOID>} <query>
[-debug] [-help]
```

This command searches for information of the collection object named in <collection LOID> or <collection path>. The format of the <query> string is described in the paper "Constructing Distributed Schedulers Using the MESSIAHS Interface Language," by Steve J. Chapin and Eugene H. Spafford.⁵

Collection queries can be constructed using the interface below.

```
<query>           è           <bool-expression>
```

```
<bool-expression>  è    true | TRUE | True |
                        false | FALSE | False
                        è    <comparison-expr>
```

<comparison-expr> è	<expr> == <expr>
è	<expr> <= <expr>
è	<expr> >= <expr>
è	<expr> < <expr>
è	<expr> > <expr>
è	<expr> <> <expr>
è	<expr> != <expr>
è	<expr> or <expr>
è	<expr> xor <expr>
è	<expr> and <expr>
è	not <expr>
è	match (<string>, <string>)

```

<expr>      è    <comparison-expr>
             è    <int-const>
             è    <oct-const>
             è    <float-const>

```

5 This paper is available on-line at <http://legion.virginia.edu/papers_abstracts.html#messiahs>

	è	<string>
	è	\$ <identifier>
	è	- <expr>
	è	~ <expr>
	è	<expr> + <expr>
	è	<expr> - <expr>
	è	max (<expr>, <expr>)
	è	min (<expr>, <expr>)
	è	<expr> / <expr>
	è	<expr> * <expr>
	è	<expr> mod <expr>
	è	int (<expr>)
	è	float (<expr>)
	è	(<expr>)
<int-const>	è	__any valid integer constant__
<oct-const>	è	0[0-7]*
<float-const>	è	__any valid float constant__
<string>	è	"."
<identifier>	è	[a-zA-Z]+

Variables are of the form \$VARNAME (e.g. \$system_arch). The official variable names will be those exported by resource objects in their attributes. To get some idea of the current set of attributes, consult the documentation on resource objects (for the most up-to-date information, invoke the `retrieve_all_attributes()` method on a resource object and examine the results).

Examples of query strings are:

```
'true'
```

This query would return the list of all objects contained in the collection.

```
'match($host_os_name, "SunOS")'
```

This query would return the list of all objects contained in the collection with an attribute of the form (host_os_name, "SunOS").

```
'match($host_os_name, "SunOS")'
```

or

```
'match($host_os_name,"Linux")'
```

This collection would return the list of all objects contained in the collection with an attribute of either (\$host_os_name,"SunOS") or (\$host_os_name,"Linux").

The following options are available:

- v** Run in verbose mode. The resulting list of objects matching the query will be displayed along with the attributes of each object. Otherwise, only the list of matching objects (without their attributes) will be displayed.
- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

The example below uses the default collection (found in the /etc context), would be:

```
$ legion_query_collection /etc/Collection true
2 hits:
1.36ab9e4a.03.01000000.000001fc099204...
1.36ab9e4a.07.01000000.000001fc0d6e07...
$
```

The output shows that there are two objects in the collection, the bootstrap host and the bootstrap vault.

legion_remove_sub_collection

```
{-l <collection LOID> | -c <collection path>}
{-l <member LOID> | -c <member path>} [-q <query>]
```

This command ends a parent-subcollection relationship. It does not destroy either collection. You must name the parent collection and the subcollection. If the parent collection is running multiple queries on the subcollection, use the -q flag to stop a specific query (use `legion_list_sub_collections`, page 50, to see which queries are running on which subcollections). Otherwise, all queries will be stopped.

The following option is supported:

- q <query>** A MESSIAHS-style query that is being run on the subcollection. See `legion_query_collection` (page 51) for details.

legion_set_default_placement

```
{[-c] <class context path> | -l <class LOID>}
{[-c] <host context path> | -l <host LOID>}
{[-c] <vault context path> | -l <vault LOID>}
[-debug] [-help]
```

Sets the default placement for objects of the class named in <class LOID> or <class context name> to the given host/vault pair. Default

placement is used when a class has no associated scheduler object or cannot contact its associated scheduler object. When a default placement is set the user should take care to ensure that an implementation for the default placement host's architecture is available.

The following options are supported:

- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

legion_set_scheduler

```
{[-c] <class context name> | -l <class LOID>}
{[-c] <Scheduler context path> | -l <Scheduler LOID>}
[-debug] [-help]
```

This commands sets the scheduler named in <scheduler LOID> or <scheduler context path> for the class named in <class LOID> or <class context path>. This will be the default scheduler for that class. The class will then use the scheduler to determine which hosts and vaults should manage its instances (i.e., determine placements for the class's instances).

The following options are supported:

- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

legion_set_scheduler_policy

```
{[-c] <class context name> | -l <class LOID>} <policy>
[-debug] [-help]
```

This command can be used to set a class object's policy for using its default scheduler. See also `legion_set_scheduler`.

The legal values for <policy> are:

- 0** This policy value specifies that the class named in <class LOID> or <class context path> should use its default scheduler if the scheduler is currently active. It is intended to protect bootstrap classes, which may be involved in activating schedulers and scheduler helper objects. Typically, this policy is not recommended for user class objects.

Classes using this policy should have a default placement available. See `legion_set_default_placement`.
- 1** This policy value specifies that the class should always use its default Scheduler, even if the Scheduler is not active. This is recommended for user classes, which usually require help from a Scheduler object when making placement decisions.

The following options are supported:

- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

legion_set_varch

```
{[-c] <host context path> | -l <host LOID>} <arch> [-debug] [-help]
```

Set a virtual host object's architecture. A virtual host object lives on host A (the physical host) but actually represents host B (the virtual host). If the `legion_set_varch` command is not run, Legion will assume that the virtual host object's architecture is the same as its physical host.

The following options are supported:

-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_set_vrun

```
{[-c] <host context path> | -l <host LOID>} <path> [-debug] [-help]
```

This command configures a virtual host object. A virtual host object lives on host A (the physical host) but actually represents host B (the virtual host). Running this command indicates where the physical host can find scripts for starting and managing jobs on the virtual host. The `<path>` parameter is the local path for those scripts.

To configure a T3E virtual host object, you could run something like this:

```
$ legion_set_vrun /hosts/my_T3E $LEGION/src/T3E/SDSC
```

This tells Legion that the scripts are in `$LEGION/src/T3E/SDSC`.

The following options are supported:

-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_vault_host_list

```
{[-c] <vault context path> | -l <vault LOID>}  
[{-a | -d | -t} <host1> <host2> ... <hostn>]  
[-p] [-debug] [-help]
```

Display and manipulates the list of hosts with which the vault named in `<vault context path>` or `<vault LOID>` can interoperate. Note that if you name the vault by its context name in the first parameter (i.e., `myVault`) you must use the hosts' context names in the `<host1>` `<host2>` ... `<hostn>` parameter. Similarly, if you name the vault by its LOID (with the `-l` flag) you must use the hosts' LOIDs.

To view the list of hosts that a given vault can operate on, you could use something like the example below.

```
$ legion_vault_host_list /vaults/VaultName -p  
** COMPATIBLE HOST LISTING:  
** 1.01.07.d49d1a40.000001fc0c04724...  
** 1.01.07.d59d1a40.000001fc094e23c...  
** 1.01.07.d69d1a40.000001fc0b68108...  
$
```

The following optional parameters are supported:

- l** Use LOIDs to specify vault and host.
- a** Add named host to the vault's acceptable host list.
- d** Delete named host from the vault's acceptable host list.
- t** Test whether or not a host is on the vault's acceptable host list.
- p** Display the vault's acceptable host list.
- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

You can use the **-a**, **-d**, and **-t** flags more than once when running the command but regardless of how you list them on the command line Legion will process them in a specific order when you run the command: first adding any new hosts, then deleting old hosts, then testing any hosts, and finally printing out the results.

2.7 General functions about the state of the system

legion_classof

```
{[-c] <object context name> | -l <object LOID>}
[-debug] [-help]
```

Displays the LOID of the class of the object named in <context path> or <object LOID>. The example below displays instance Foo's class LOID.

```
$ legion_classof Foo
The class of 1.01.66000000.01000000.000001fc0... is
1.01.66000000..000001fc0d085b2c33...
$
```

The following options are supported:

-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_create_stat_tree

```
<base context path> [-debug] [-help]
```

Creates a database of current loads on all host objects in a system.

The following options are supported:

-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_get_accounting_data

```
[-v] [-help] [-debug]
{<pull-source1> <pull-source2> ... <pull-sourcen>}
{-d <local target directory>}
[-ot <seconds>] [-it <seconds>]
```

This command retrieves accounting information logs from one or more specified pull-sources and places them in the specified target directory in your local file space. A *pull-source* can be an individual host, a list of hosts, a class of hosts, or a Unix file containing a list of Legion hosts. Each host is queried for its log of process/job accounting information.

The host object(s)'s current status is also retrieved and placed in the target directory as a log file. If all the requested accounting information is successfully gathered, the queried hosts are instructed to trim their logs.

You must include the following parameters.

```
<pull-source1> <pull-source2> ... <pull-sourcen>
```

One or more hosts that should be queried for account information. A pull-source can be one of the following:

- c** <host context path>
- l** <host LOID>

- Class** <host class context name>
- ClassLoid** <host class Loid>
- Context** <context containing a group of hosts>
- File** <local Unix file>

If you use the **-File** flag, the specified file must contain a list of Legion host context paths or LOIDs.

-d <target directory>

A local directory that will contain the accounting logs.

The following options are supported.

- v** Run command in verbose mode. This will return information about which log requests are currently being processed.
- help** Print command syntax and exit.
- debug** Catch and print Legion exceptions.
- ot** <seconds> Outcall timeout in seconds (default is 60 seconds).
- it** <seconds> Incall timeout in seconds (default is 60 seconds).

legion_host_stats

```
{[-c] <host context path> | -l <host Loid>}
[-debug] [-help]
```

Prints the number of objects and current load for the given host object. The following options are supported:

- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

legion_list_objects

```
{[-c] <host context path> | -l <host Loid>}
[-debug] [-help]
```

Lists the Legion objects currently managed by (i.e., running on) the host object named in <host context path> or <host Loid>. This may include each object's Loid, status, and its owner's Loid and OA.

The following options are supported:

- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

legion_update_accounting_db

```
[-v] <local configuration file path> <local logfile directory>
```

Parse the data in all accounting information logs in a specified local directory and place the data into a MySQL database. This tool requires the following system configurations in order to run properly:

- Linux/Unix variant operating system
- Perl version 5.005_05 or higher
- Access to a MySQL database whose scheme has been configured for Legion accounting information
- Msql-Mysql-modules and Data-Dumper Perl modules installed

The tool may attempt to perform reverse context lookups on object LOIDs in order to determine human readable identifiers (i.e., context paths). For suitable results, we suggest that you only run this command if you are logged in as /users/admin.

You must include the following parameters.

<local configuration file path>

A local text file that contains database configuration information in this format:

```
[client]
host=
database=
user=
password=
```

<local logfile directory>

A local directory that contains accounting information logs.

The following option is supported.

- v** Run in verbose mode. Print information about which log requests the command is currently working on.

legion_update_stat_tree

```
[-v] [-help] [-t <repeat delay seconds>]
[-site <top site context path>]
[-collection <collection context path>]
```

Updates a database of current loads on all host objects in a system. You can set the command to run at regular intervals.

The following options are supported:

- v** Verbose mode.
- help** Print this help message and exit.
- t** Specify that the command should operate in continuous mode, continually updating the stat tree every <repeat delay seconds> seconds. Use Ctrl-C to quit out of continuous mode.
- site <site context path>** Specify the top-level context to update. The default starting point is /sites.
- collection <collection context path>** Specify a collection to satisfy host information requests. The default is /etc/DefaultCollection.

```
legion_version
  [-debug] [-help]
```

Prints your Legion version number. For example, the output below indicates that the current system is University of Virginia Legion version 1.5.9.

```
$ legion_version
VaL Legion version 1.5.9
$
```

The following options are supported:

-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

```
legion_wellknown_class
  <wellknown class name>
  [-debug] [-help]
```

Gets the class LOID of a Legion "well-known" class. Possible values for <wellknown class> are:

- LegionClass
- BootstrapMetaClass
- UnixHostClass
- UnixVaultClass
- CommandLineClass
- ContextClass
- ImplementationClass
- UnixImplementationCacheClass
- BindingAgentClass

The following options are supported:

-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

```
legion_whereis
  {[-c] <object context name> | -l <object LOID>}
  [-debug] [-help]
```

This command returns a specified object's host and vault object. That is, the output gives the context path of the host and vault objects where the object and the object's persistent state are located. If the host or vault objects have been assigned aliases Legion will randomly return one of the context paths. Your output will look something like this:

```
$ legion_whereis Foo
Host: /hosts/BootstrapHost
Vault: /vaults/BootstrapVault
$
```

This example looks up the location of object Foo. The output shows that the object is located on `BootstrapHost` and its persistent state is on `BootstrapVault`. The following options are supported:

- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

`legion_whoami`
`[-debug] [-help]`

You can use this command to find out which user id you are currently logged in with. For example, if you log in as user nemo, your output will be:

```
$ legion_whoami
/users/nemo
$
```

The following options are supported:

- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

2.8 Security

`legion_add_acl`

```
[[ -c ] <object context path> | -l <object LOID>] { -s | <filename> }
[ -debug ] [ -help ]
```

Adds an access control list to the object named in `<object LOID>` or `<object context path>` or, if no object is named, the current environment. This command resembles `legion_set_acl`, but it merges in a new access control list. For example, if the current access control set contains access control lists for methods A and B of class XYZ's instances, adding new access control lists for class XYZ's methods B and C will not change A, replace B, and add C.

The following options are supported:

-s	Read from stdin.
-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

`legion_add_implicit_params`

```
[ -l <AuthenticationObject LOID> |
  -c <AuthenticationObject context path> ]
{ -s | <filename> }
[ -debug ] [ -help ]
```

Add an implicit parameter to the `AuthenticationObject` named in `<AuthenticationObject LOID>` or `<AuthenticationObject context path>` or, if no `AuthenticationObject` is named, the current environment. The arguments take the same input format as `legion_set_implicit_params`. New parameters merge into the existing implicit parameter set (i.e., new parameters override old ones of the same name). There is no way to remove or unset selected implicit parameters in the `AuthenticationObject` (please see `legion_modify_parameters` on page 69 for information on changing your current session's implicit parameters).

The following option is supported:

-s	Read from stdin.
-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

`legion_change_acl`

```
{ [ -c ] <object context path> | -l <object LOID> }
<function pattern> <modifications>
```

Change an access control list (ACL) for a set of functions on a specified Legion object.

Please note that if you have never altered the specified object's ACL before the object's permissions are by default set for Empty Function. In this setting, only the object's owner, the `admin` user, and the object's

class can call the function. When you change the object's permissions for one or more functions, the set of allowed or denied users is merged with the Empty Function list. That is, if you add user John to the Allowed list for object Foo's `getInterface()`, the list will now contain four users: you (the owner), admin, Foo's class, and John.

The following parameters are required:

<function pattern> Lets you specify a Unix-like wildcard argument that indicates which functions should be modified. Can contain any standard wildcard pattern (e.g., *, ?, [], and ^).

<modifications> A list of statements that add/remove users or groups from the specified functions for the specified Legion object. It can contain these elements:

+allow | -allow | +deny | -deny

{-l <user group LOID> |

[-c] <user group context path>}

Legion checks the Denied list before it checks Allowed, so if a user or group is on both lists, Legion will consider it denied.

You can list multiple <modifications> in a single run. For example, you can alter SomeFile's ACL to allow SomeUser and SomeGroup to call any functions with "read" in them (such as `read_hosts_contexts()` or `get_ready_time()`):

```
$ legion_change_acl /home/legion/SomeFile "**read*" \
+allow /users/SomeUser +allow /users/SomeGroup
```

Note that you need full context paths for objects in the <modifications> parameter.

The next example changes SomeContext's ACL. It denies SomeUser and allows SomeGroup the right to call any function with "multiLookup" in it.

```
$ legion_change_acl /home/legion/SomeContext \
"**multiLookup*" +deny /users/SomeUser \
+allow /users/SomeGroup
```

legion_change_owner

[-v] [-r]

{[-c] <object context name> | -l <object LOID>}

{-l <target owner LOID> | -c <target owner context path>}

[-debug] [-help]

Changes an object's owner. This command currently works only on unclaimed objects: if an object is already owned you would have to be

logged in as both the current owner and the target owner in order to run this command.

The following options are supported:

- v** Run this command in verbose mode.
- r** Run this command in recursive mode. If the specified <object LOID> or <object context path> is a class, ownership of all instances, sub-instances, etc. will change. If the specified <object LOID> or <object context path> refers to a context object, change ownership of all context entries, recursively applying the operation to sub-contexts will change. In either case, ownership of the root object referred to by <object LOID> or <object context path> is changed.
- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

legion_change_permissions

```
[+-rwx] [-v] <group/user context path>
<target context path>
[-debug] [-help]
```

Changes an object's read, write, and execute permissions so that you can allow the user or group named in <group/user context path> to use the object named in <target context path>. This tool manipulates an object's access control list (ACL) so that other users can call methods on your objects.

For our purposes, "read" methods are defined as methods that obtain but do not modify an object's state, "write" methods are methods that modify an object's state, and "execute" methods are methods that run an object. For example, giving bob "read" permissions for object foo, below, lets him view but not alter foo's state.

```
$ legion_change_permissions +r /users/bob foo
```

This command works on common Legion object types: context, file, class, tty, implementation, host, and vault objects all fall into this category.

The following optional parameters are supported:

- r** Deny read permissions to the target object.
- +r** Grant read permissions to the target object.
- w** Deny write permissions to the target object.
- +w** Grant write permissions to the target object.
- x** Deny execute permissions to the target object (this option is for class objects only).

+x	Grant execute permissions to the target object (this option is for class objects only).
-v	Run the command in verbose mode. You will see a list of the ACLs that have changed.
-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_create_user

```
[-a <admin path>] [-z <password>]
[-h <new home context>] [-c <AuthenticationClass path>]
<user id path>
[-debug] [-help]
```

This command is actually a simple wrapper around the `legion_create_user_object` command. The full command give more control to the creation of AuthenticationObjects.

The user id is the context name of an AuthenticationObject: the `legion_create_object` utility creates the object and assigns it the contextname given in `<user id path>`. Please note that this argument is a full path (relative or absolute). Note also that the user id's context has nothing to do with that user's privileges in that context. Once a user is created, the `legion_login` command is used to log in. The command will prompt for a password for the new user (unless you use the `-z` option), and will return the new object's LOID.

The following options are supported:

-a <admin path>	Specify the admin object's context path. Default is <code>/users/admin</code> .
-z <password>	Specify the new user id's password. The default setting will prompt for the password after the command has been run. This option is not recommended for casual use, since the password will be visible on the command line.
-h <new home context>	Specify the new user's home context. Default is <code>/home/<user id></code> .
-c <AuthenticationClass path>	Specify the AuthenticationClass which you wish to instantiate. Default is <code>/class/AuthenticationClass</code> .
-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_create_user_object

```
{[-c] <class context name> | -l <class LOID>}
[-h <host for new object> | -v <vault for new object>]
[-f <implicit parameter file>] [-z <password>] <user name>
[-debug] [-help]
```

Creates a new object to represent a new Legion user id. The new object can be an instance of the AuthenticationObjectClass or another class that implements AuthenticationObjects.

The following options are supported:

- h** <host for new object> Specify where the new object should be created.
- v** <vault for new object> Specify where the new object's persistent state should be stored.
- f** <implicit parameters filename>

This option names a file containing implicit parameters to be stored in the newly created AuthenticationObject (handed out on legion_login). The filename can be "-", in which case the command reads the parameters from the standard input.

Note that these are *not* the implicit parameters that control the behavior of the new AuthenticationObject. Upon login, the user's environment will be set to contain the latter set of implicit parameters, and they will affect the creation of all subsequent objects.

This option is intended to make it easier to administer the system. In particular, admin can create user accounts that include implicit parameters that by default give the admin access rights to all of the users' subsequently created objects.

- z** <password>

Include the new user id's password in the command line. This option is not recommended for casual use, since the password can potentially be seen by other users while the command is operating.

- debug**
- help**

Catch and print Legion exceptions.
Print command syntax and exit.

legion_get_acl

[[**-c**] <object context path> | **-l** <object LOID>]
[**-debug**] [**-help**]

Returns the access control list of the object named in <object LOID> or <object context path>, or, if no object is named, the default access control set associated with the current logged-in Legion session. This command returns 2 as its exit status if the object/session has no access control list.

The following options are supported:

-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_get_implicit_params

[[**-c**] <object context path> | **-l** <object LOID>]
[**-debug**] [**-help**]

Get the implicit parameters of the object named in <object LOID> or <object context path>, or, if no object is named, the current logged-in Legion session. The former is analogous to looking in a Unix .profile or .cshrc for which environment variables get set at login and the latter is analogous to Unix printenv.

The following options are supported:

-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_init_Apps

Initializes the Applications package, which extends the basic Legion package. Please see page 9 in the System Administrator Manual for more information on the Legion packages.

legion_init_Extra

Initializes the Extra package, which adds functionality to the core Legion system. Please see page 9 in the System Administrator Manual for more information on the Legion packages.

legion_init_HPC

Initializes the High Performance Computing package, which lets you run programs in Legion. Please see page 9 in the System Administrator Manual for more information on the Legion packages.

legion_init_security

Creates the initial user (called by default /users/admin) in a new Legion system. This command should be run immediately after `legion_initialize` when you are starting your system. The script creates a new context called /users, a new user object called admin. The context name admin is placed in the new /users context. You will be asked for a password.

After running `legion_init_security` you must login as admin in order to use the system. Use the `legion_login` command, with `/users/admin` as the `<user name>` parameter. This command only needs to be run once, when the system is first started. There is only one `/users/admin` in a system, since it has system administrator-like privileges. You can change the admin's password later with the `legion_passwd` command, if you wish.

`legion_login`

```
[<user id path> | -l <user id LOID>] [-p <password>]  
[-debug] [-help]
```

Allows user to log in to the Legion system as a user, and sets up implicit parameters and security credentials for the user. Note that the command can be run without any arguments (although it will prompt for a user name if a user LOID or user id is not given). User names are context names for AuthenticationObjects (special objects that contain a user password, initial implicit parameters, and other information), created with the `legion_create_user` command.

On a successful login, a credentials file (a user read-only file) is created in your local `/tmp` directory. The user's shell must be able to see this file so that his/her command-line utilities can use it. The file will be removed when the user logs out with `legion_logout` (below). You get a separate credentials file for each shell in which you run `legion_login`.

The following options are supported:

- p <password>** Enter the password from the command line. **This method is not secure.**
- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

`legion_logout`

```
[-debug] [-help]
```

This logs out a user out of Legion. This command removes the credentials file that is created by the `legion_login` utility and has the effect of logging the user out of a secure Legion system.

We strongly recommend that users insert this command into a `.logout` file or some other script that is run when a user logs out. This ensures that credential files do not remain in the system for unnecessarily long periods.

The following options are supported:

- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

legion_modify_parameters

[-debug] [-help] <argument>

Allows a user to modify the implicit parameter set for his or her current session. These parameters can include anything the user wishes to place in them but usually include the user's credentials obtained when he or she logged in to their Authentication object.

The <argument> parameter must contain one of the following:

- l**
- p** [<param type>] <param name>
- d** [<param type>] <param name> [<param value>]
- a** <param type> <param name> <param value>

If you use -p, -d, or -a in your <argument>, <param type> must be one of the following values:

STRING_PARM	A string parameter.
FUNCTIONID_PARM	A parameter describing a function identifier.
INT_PARM	An implicit parameter containing an integer value.
LOID_PARM	An implicit parameter containing a valid LOID.

The following options are supported:

- l** List all of the implicit parameters in the user's current Legion session.
- p** Print a specific set of implicit parameters in the user's current Legion session.
- d** Delete a specific implicit parameter (or a set) from the user's current Legion session.
- a** Add a new implicit parameter to the user's current Legion session.
- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

legion_passwd

{<user id context> | -l<user id LOID>}
[-debug] [-help]

Changes a user's password: the command will prompt for the new password. You must be logged in as either admin or the user in order to successfully run this command. You must provide the full context path for the user id (e.g., /user/<user id>).

The following options are supported:

- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

legion_set_acl

```
{[-c] <object context name> | -l <object LOID>}
[-s | <filename>]
[-debug] [-help]
```

Sets the access control list of the Legion object named in <object context path> or <object LOID>. The default will set the access control set for the current environment. The input file should be an implicit parameters file (see \$LEGION/src/UserObjects/Security/SampleImplicitParams for an example). The implicit parameters file will be scanned only for access control information pertaining to the specified object or environment; all other entries in the file will be ignored.

This access control list is inherited by any newly created objects in the current login session (it roughly corresponds to a Unix umask).

The following option is supported:

-s	Read from standard input.
-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_set_implicit_params

```
{[-c] <object context name> | -l <object LOID>}
[-s | <filename>]
[-debug] [-help]
```

Set the implicit parameters of a specified AuthenticationObject or, if no object is named, the current environment. Any parameters that were previously set are cleared. These are the parameters that the user inherits when he logs into Legion. They are passed along as different Legion commands are executed and are similar to environment variables that user might set up in a Unix .profile or .cshrc file.

If no AuthenticationObject is specified, the current Legion login session's implicit parameters are modified (the changes will not persist to the next login session). The command reads the implicit parameters from a file (or the standard input). A sample file can be found in \$LEGION/src/UserObjects/Security/SampleImplicitParams.

The following option is supported:

-s	Read from standard input.
-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_set_message_security

```
[-c <AuthenticationObject context path> |
-l <AuthenticationObject LOID>]
{Off | Protected | Private} [-help]
```

This command updates the message layer security setting of an AuthenticationObject or the current environment by changing the "MessageSecurity" implicit parameter. There are three possible

settings: Off, Protected, and Private. The setting determines the object's message encryption level. The default setting is Protected.

The arguments take the same input format as `legion_add_implicit_params` (page 62). New parameters merge into the existing implicit parameter set (i.e., new parameters override old ones of the same name).

You must include one of the following parameters:

Off	No encryption or digesting of messages
Protected	Message digested for verification and credentials encrypted for authentication
Private	Message and credentials encrypted for authentication and verification

The following option is supported:

-help	Print command syntax and exit
--------------	-------------------------------

`legion_show_acl`

```
{[-c] <object context path> | -l <object LOID>}
[<function pattern>] [-showLoids]
```

Display an object's access control list (ACL) in human-readable form. The optional `<function pattern>` argument allows you to use Unix-like wildcard arguments to control which functions are listed. The output may list one or more users as Unknown. This means that you do not have permission to see that user(s)'s context name.

The following options are supported:

<code><function pattern></code>	Lets you use a Unix-like wildcard argument to list specific functions. You can use any of the standard wildcard patterns (*, ?, [], and [^]).
-showLoids	If Legion is unable to display a user's or group's context path, it will print a LOID (instead of unknown).

For example, if you asked to see object `/home/spw4s`'s ACL for all functions containing "MayI" in their names, you would see something like this:

```
$ legion_show_acl /home/legion "**MayI*"
Access to Function "_4MayI_6GetACL":
  Allowed:
    /users
    <Unknown>
    /users/legion
  Denied:

Access to Function "_4MayI_6SetACL_23LRef<AccessControlList>":
  Allowed:
    <Unknown>
    /users/legion
  Denied:
```

```

    Access to Function
    "_4MayI_17LegionCheckAccess_30LRef<LegionFunctionIdentifier>23LRef<A
    ccessControllist>26LRef<LegionCertificateSet>i":
        Allowed:
            Empty
        Denied:
$

```

The results list each function's Allowed and Denied list. Notice that the third function's Allowed list is "Empty." This means that it contains an Empty Function set, the default setting for all Legion object functions. The Empty Function set contains the object's owner, the admin user, and the object's class. These three objects are automatically given permission to call all of the object's functions. You can use `legion_change_acl` to change these lists (page 62).

2.9 Application development

legion_bfs

```
<file> [-o <out file>] [-BackEnd]
[-help]
```

Parses Fortran programs containing Legion Basic Fortran Support (BFS) pseudo-comments to produce standard Fortran output, or parses Legion BFS IDL files to produce Mentat Programming Language (MPL) output. If the input <file> ends with .f, it is treated as Fortran with pseudo-comments. If the input <file> ends with .bfs, it is treated as BFS IDL.

The following options are used:

-o <out file>	Name of the file containing the resulting Fortran or MPL output.
-BackEnd	Instructs the legion_bfs compiler to produce output for IDL files included in the Fortran input file (ignored when the input is a BFS IDL file).
-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_make

```
[-v] [-a <architecture>] [-h <host context path>]
[-e <make command>] [-OUT <remote file>]
[<arg1> <arg2> ... <argn>]
[-debug] [-help]
```

Compiles programs on a remote host. The command tars your current directory, copies it to a remote host, untars it, and executes a make command. The resulting compiled binaries can be registered with Legion via the directory's makefile or can be copied back to your current directory with one or more -OUT options. Be aware that legion_make will tar everything in and below your current directory, so be sure that you are in the correct directory before running it.

If no remote host or architecture is specified (with the -h or -A flags) the command will run on a random host. For example, the following command would build the tar file on a random Alpha Linux host:

```
$ legion_make -a alpha_linux
```

To be more specific, use the -h flag:

```
$ legion_make -h /hosts/HostFoo
```

The following options are supported:

-v	Run in verbose mode (up to four of these flags may be used).
-----------	--

-a <architecture>	Specify what kind of platform should be used to execute the command.
-h <host context path>	Specify which host should execute the program.
-e <make command>	Specify an executable command (other than make) to be run on the remote host.
-OUT <result file>	Specifies the local path name of a file that should be copied out of the remote program's current working directory after the program terminates. The default setting does not copy anything back to your current directory.
<arg1> <arg2> ... <argn>	Optional arguments passed to the make commands: they can specify make targets or other build parameters.
-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_make_multi

```
[-v] [-a <architecture>] [-e <make command>]
[<arg1> <arg2> ... <argn>]
[-debug] [-help]
```

Compile a program on multiple platforms. Like the `legion_make` command, this tars the current directory, copies it to a remote host, untars it, and executes a make command on it. The resulting compiled binaries can be registered with Legion via the directory's makefile. Optional flags let you specify different architectures and make commands.

The following options are supported:

-v	Verbose mode. Print actions as command executes.
-a <architecture>	Specify an architecture on which to run make. This option can be specified multiple times to request concurrent builds on multiple platforms. If no architectures are specified, a remote make for the current architecture (\$LEGION_ARCH) is performed.
-e <make command>	Specify the make-command for the remote build (default is make).
<arg1> <arg2> ... <argn>	These optional arguments are passed to the remote "make" command. They can specify make targets or other build parameters.
-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

`legion_mplc`
 [`<flags>`]

Compiles MPL (Mentat Programming Language) programs. There are several possible parameters to this command: please see the MPL Manual for more information (the MPL Manual is available from the Legion web site at [<http://legion.virginia.edu/documentation.html>](http://legion.virginia.edu/documentation.html)).

`legion_mplc_reg_impl`
`<class name> <binary path>`
`<stateless | stateful | sequential> <arch>`
 [`-help`]

Registers the given binary, named in `<binary path>`, as a stateful, stateless, or sequential Mentat object for the architecture named in `<arch>` and under the class object named in `<class name>`. This command is automatically run when you use the `legion_mplc` command (without the `-nolegion` option) to compile an MPL program.

In the example below, `MentatObject` is a stateful Mentat object in the local path `/home/legion/bin` on a `linux` or `alpha_linux` machine. To register it as belonging to a class with the context path name `/ClassObject/MentatObject` in `linux` or `alpha_linux` architectures, you would enter:

```
$ legion_mplc_reg_impl /ClassDirectory/MentatObject \
/home/legion/bin/MentatObject stateful linux
```

or

```
$ legion_mplc_reg_impl /ClassDirectory/MentatObject \
/home/legion/bin/MentatObject stateful alpha_linux
```

The following option is supported:

-help Print command syntax and exit.

`legion_output_state`
`{[-c] <object context name> | -l <object LOID>} | StartupState`
 [`-debug`] [`-help`]

This command uses a special output device for debugging and information reporting. When output is sent using this mechanism, the object(s) will decide whether that type of information is to be displayed and where to display it. Users will be given a series of options to selectively filter what output is displayed to the screen. The output levels include:

- Warnings
- Information
- Low Level Debug
- High Level Debug
- Errors
- Fatal Errors

There are two ways to use this command. You can dynamically change which of a specified object's output types are filtered out and which are displayed. This option also allows the user to change the location of the output (from stderr, stdout, and any Unix file they want).

Or, you can use the `StartupState` option, in which case the command will print instructions for bringing up a new system with the desired output state.

Please note: if you use `StartupState`, `legion_output_state` does not contact Legion and therefore does not run as a Legion program. I.e., you can execute it outside of a running Legion system. Otherwise, it runs as a normal Legion command-line tool.

The following options are supported:

- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

`legion_record`

```
{-uf <local storage file name> | [-c] <recorder context path>}
[-name <debug session name>] [-t <interval>]
<client application> [<arg1> <arg2> ... <argN>]
[-debug] [-help]
```

Executes the application named in `<client application>` in record mode so that it can later be played back with the `legion_replay` tool. All objects started by the application will record all relevant event activity in a local file, named in `<local storage file name>`, or in a previously started Legion recorder object, named in `<recorder context path>` (see below). The utility also monitors the application and reports if it dies. Any application can be started with `legion_record`, but the command is most useful for Legion applications.

Only one execution of the application will be recorded in the storage file at a time. If you record again to the same storage file you will overwrite previous data.

If you wish to use a Legion recorder object to record your application's activity, you must start the recorder object separately (i.e., `legion_create_object -c /class/RecorderObject`).

If you wish to record a server object (i.e., one that is not created by the client application, but instead exists on its own), you must start the server object in record mode independently of the client application.

The following options are supported:

- name <debug session name>** Specify a name for the debug session. If this option is not used, a random name will be chosen.
- t <interval>** Specifies how long (in seconds) the `legion_record` utility should wait between health checks. These health checks are used to poll the

recorder object to learn about object death. The default setting is 30 seconds.

-debug

Catch and print Legion exceptions.

-help

Print command syntax and exit.

legion_replay

```
{-uf <local storage file> |
  [-c] <recorder context path> -name <debug session name>}
{-list | [-cmd <debugger>] [-local] <session number>}
[-debug] [-help]
```

Begins a replay or playback session for a recorded object. This command is used in tandem with the `legion_record` tool to debug a particular application. The `legion_replay` tool starts a debugging session for an object started in the application. If used with the `-list` option, the command returns a list of the sessions stored in the storage file, the session number for each object that was started, and the final states of the objects. These session numbers can then be used to debug the objects one at a time, by rerunning the command with the correct session number.

You must use one of the following sets of parameters to specify which session you wish to start. Sessions are named in `legion_record` (see above). If you used a Legion recorder object, you'll need to provide the debug session name as well.

The following options are supported:

-uf <local storage file>	The local file that was used to record your session.
-c <recorder context path>	The Legion recorder object that was used to record your session (must be accompanied by debug session name).
-name <debug session name>	The desired debug session name.
-list	Display the sessions in the storage file.
-cmd <debugger>	Specify a debugging program (e.g. <code>gdb</code> , <code>xdb</code> , etc.) to run on the storage file. Default is <code>gdb</code> .
-local	Indicates that the debugging session should be replayed on your local machine. The default will start a remote session on the machine which executed the original object.
-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_stateless_add_workers

```
<class name> <worker name1> <worker name2> ... <worker nameN>
[-debug] [-help]
```

Creates additional MPL stateless object workers. By default, when a stateless class is created or registered in context space it instantiates a single worker. That worker is used by all other objects wishing to call a remote method invocation on that class. Adding more workers allows message calls to be handled more efficiently.

The <class name> parameter should be the stateless class object's context name. The <worker name> parameters should be the desired context name of the new worker objects. Note that you can use full or relative path names.

The following options are supported:

- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

legion_stateless_configure

```
<stateless class context path> [-n <number of replicas>]
[-Ch <host context path>] [-w <max number work requests>]
[-FLUSH]
[-debug] [-help]
```

This command configures stateless Legion classes so that requests sent to a stateless class are first routed to a proxy object, which selects a worker to service the requests. You can use flags to control the number of replica workers, their placement, and the number of work requests that a worker may service at a given time.

The current version of the proxy object uses a self-scheduling algorithm. When N work requests have been farmed out, the proxy will store any pending requests in an internal queue: when a worker is available, the proxy assigns it a request from the queue.

N here is given by the following formula:

$$N = \text{<number of replicas>} * \text{<max number of workers>}$$

The following options are supported:

- n <number of replicas>** Specify the number of workers to be used. Default is 1.
- Ch <context containing list of hosts>**
Name a host where the work requests should be executed.
- w <max number of work requests>**
Specify the maximum number of work requests that each worker should service at a time. Default is 5.
- FLUSH** Remove all work requests for this class. You must spell this flag as written with proper capitalization.

-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

legion_stateless_remove_workers

<class name> <worker name1> <worker name2> ... <worker nameN>

[-debug] [-help]

Removes MPL stateless object workers. The <class name> parameter should be the stateless class object's context name. The <worker name> parameters should be the context name of the unwanted worker objects. Note that you can use full or relative path names.

The following options are supported:

-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

2.10 Program support

legion_ft_initialize

This is a script that initializes the CheckpointStorage object. This object is used by MPI fault tolerance implementation and must be run before you can use the `legion_mpi_run` fault-tolerant flags (page 90). You only need to run it once, however.

legion_FTPd

```
[-help] [-v] [-p <port number>] [-a] [-m <max connections>]
[-t <transfer block size>] [-o <max outstanding invocations>] [-d]
[-w] [-e] [-rt <RMI timeout>] [-dt <detail timeout>]
```

Starts the Legion FTP daemon from the command line. This daemon is a Legion-enabled server tool for accessing Legion context space via FTP. Once you've started the daemon, you can use a standard FTP client software to view, store, and retrieve objects from your context space.

Once started, the daemon's operating parameters cannot be changed. To terminate, send a SIGINT or `^ - c` signal.

Please be aware that if you use the `-a` flag, other users will be able to use your daemon to view and use your context space. Your password will also travel in the clear over the network.

The Legion FTP daemon will only work for systems running Legion version 1.7 or higher. It implements the minimum requirements of an FTP server as outlined in RFC-959 and RFC-1123.

The following options are supported:

- | | |
|---|--|
| -help | Print the usage information and default values for selected parameters and exit. |
| -v | Operate in verbose mode. All commands and responses will be echoed to stdout |
| -p <port number> | Specify a listening port for incoming FTP connections. |
| -a | All machines other than the local host to connect to the daemon. |
| -m <max connections> | Specify the maximum number of simultaneous FTP connections. |
| -t <transfer block size> | Specify the block size in bytes for Legion file object invocations. |
| -o <max outstanding invocations> | Specify the maximum number of outstanding Legion remote method invocations. If the <code>-d</code> flag is also enabled, only <code><max outstanding</code> |

	invocations> will simultaneously be queried for details. This flag also restricts the number of simultaneous file transfer block requests/sends.
-d	Specify that object details be retrieved during directory listings.
-w	If the -d flag has been enabled, Unix-style world access rights (r/w/x) will be displaced during directory listings. The SITE CHMOD command will also be enabled, allowing modification of individual object's world rights.
-e	Encrypt all Legion communications. If you set this option and do not enable the -a flag you will have the highest level of data security.
-rt <RMI timeout>	Specify the Legion remote method invocation (RMI) time out value in seconds.
-dt <detail timeout>	Specify the time out value for looking up object details (use in conjunction with the -d flag).

legion_link

```
[-CC <compiler>] [-Fortran]
[-FC <compiler>] [-pvm] [-mpi]
[-L<library path>] [-l<library>]
[-v] [-o <output file>]
[-bfs] <object file list>
[-debug] [-help]
```

This command links together a set of object files and libraries to produce an executable program and automatically binds the produced executables to the Legion libraries. It is similar to the Unix `ld` command. It can be used with object files and libraries created from C, C++, or Fortran source code.

If any Fortran object files are linked, the `-Fortran` flag must be included. If you specify a Fortran compiler with the `-FC` flag, the `-Fortran` flag is automatically implied.

The following options are available with this command:

-CC <compiler>	Select a C++ compiler to perform linkage. The default value is based on the default compiler used by the Legion system on your platform.
-Fortran	This flag must be included if any Fortran object files are linked.
-FC <compiler>	Select a Fortran compiler. This flag implies the <code>-Fortran</code> flag, but the two can be used together. Currently supported <compiler>

options are:

- fort Compaq Fortran compiler (Alpha Linux only)
- g77 Gnu g77
- pgf77 Portland Group pgf77 compiler (Intel Linux)
- pgf90 Portland Group pgf90 compiler (Intel Linux)
- pvm** Link the produced executable to the Legion PVM compatibility library.
- mpi** Link the produced executable to the Legion MPI compatibility library.
- L<library path>** Include the <library path> in the set of directories searched for libraries. There is no space between the "-L" and the library path.
- l<library>** Link against the specified library. There is no space between the "-l" and the library name.
- v** Provide a verbose output as the command is running.
- o <output path>** Specify the local path of the resulting program. The default is a.out.
- bfs** Link the produced executable to the Legion Basic Fortran Support (BFS) library.
- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.

legion_manage_job

[-help] [-k] [-cp <priority>]
[-fr] [-nc] [-q <context name>] <ticket number>

This tool manages a job started with the `legion_nq` (page 93) command-line tool. You must provide the job's ticket number. If the job has finished or failed, `legion_manage_job` will by default copy back output files, clean up temporary files, and mark the job's record for deletion. Use `-nc` to override this.

Use `-fr` to force the job to restart. Only the job's owner and the system administrator can perform these operations.

The following parameter is used with this command:

<ticket number> The job ticket number (assigned when you ran `legion_nq`, page 93)

The following options are supported:

- help** Print this help message.
- k** Kill the job (job owner and system administrators only).
- cp <priority>** Change the job's priority.

- fr** Force the job to restart (job owner and system administrators only).
- nc** No file clean up.
- q <context path>** Specify a JobQueue object. Default is /etc/JobQueue.

legion_manage_queue
 [-help] [-purge]
 [-maxrs <slots>]

[-q <context name>]

Use this command to manage a Legion JobQueue object. If you run it without flags, you will see all jobs that the object is currently managing. If you are a system administrator, you can also use the **-maxrs** flag to set the maximum number of job slots and the **-purge** flag to kill all of the object's jobs.

The following options are supported:

- help** Print this help message.
- purge** Kill all jobs/reset queue (system administrators only).
- maxrs <slots>** Set number of total run slots (system administrators only).
- q <context path>** Specify queue object. Default is /etc/JobQueue.

legion_mpi_debug
 [-q] {[-c] <program instance context>}
 [-help]

A utility program that allows the user to examine the state of MPI objects and print out their message queues. The **<program instance context>** is the context that is holding your program's instances. Normally it is /mpi/instances/<program_name> (/home/<user_name>/mpi/instances/<program_name> in a secure system) unless you specified otherwise when you started the program (i.e., if you used the **-p** flag when you ran **legion_mpi_run**). This command will return a list of all of the program's instances and what each one is doing. This is a handy way to debug a deadlock.

The following option is available with this command:

- q** List the contents of the queues.
- help** Print command syntax and exit.

If you use **-q**, the command will return a list of instances that are waiting to be received by the queues.

There are a few limitations: if an MPI object doesn't respond, the command will hang and not go on to query additional objects. An MPI object can only respond when it enters the MPI library; if it is in an endless computational loop, it will never reply. Output goes to stdout.

legion_mpi_probe

```

[-help] [-debug] [-v[erbose]]
[-all] [-list]
[-in <context path name>] [-out <context path name>]
[-IN <local file name>] [-OUT <local file name>]
[-showscratch] [-stat <remote file name>]
<pid context name>

```

Checks an MPI program started on a remote host with `legion_mpi_run` (page 85). This command resembles `legion_probe_run` (page 96) in that you can check on jobs, move input and output files between your local and execution hosts. It doesn't use a probe file, however, since you can contact your job through its pid context.

The following parameter must be used:

<pid context name> The context that contains the MPI instance LOIDs associated with the job.

The following options are supported:

- help** Print command syntax and exit.
- debug** Run in debug mode.
- v[erbose]** Run in verbose mode.
- all** Perform operation on all instances (default is instance 0).
- list** List all files in the remote program's working directory.
- in <context path name>** Context path of a Legion file object whose contents should be copied into the remote program's current working directory after execution begins. The local file will have the same name as the `<context path name>` basename.
- out <context path name>** Context path of a Legion file object whose contents should be copied out of the remote program's current working directory after the program terminates. The local source file will have the same name as the `<context path name>` base-name. This tool will not check to see if the job has finished, so it may copy out an incomplete file.
- IN <local file name>** Similar to the `-in` but operates on a file in the `legion_mpi_probe` local execution environment (i.e., the file named in `<local path name>`).
- OUT <local file name>** Similar to the `-out` option but operates on a file in `legion_mpi_probe`'s local

execution environment (i.e., the file named in <local path name>).

-stat <remote file name>

Print the status of a particular file in the remote job's present working directory.

-showscratch

Print the archive context scratch space. If the run hasn't yet finished or if `legion_mpi_run` dies or is killed before the job finishes, this space will contain -OUT files that haven't yet been copied out.

`legion_mpi_register`

<class name> <binary local path name> <platform type>
[-help]

MPI implementation generally require that MPI executables reside in a given place on a disk. Legion's implementation objects have a similar requirement. This command registers the executables of different architectures for use with Legion's MPI. It creates MPI-specific contexts, a class object, and an implementation for the program, and registers the name in Legion context space. If Legion security has been enabled the newly created contexts will be placed in the user's `/home/<user_name>/mpi` context *except* if the user is `admin` or not logged in. If Legion security has not been enabled, the newly created contexts will be placed in the `/mpi` context.

The command can be executed several times, if you have compiled your program on several architecture.

The following option is supported:

-help

Print command syntax and exit.

`legion_mpi_run`

{-f <option file> [<flags>]}
| {-n <number of processors> [<flags>] <program class>
 [<arg1> <arg2>... <argn>]}

Starts an MPI program. The <program class> argument requires the full context path for the class (the class is created through `legion_mpi_register`: if you have not previously registered the program, you'll need to do so before you can run it). Parameters used for this command are:

-f <option file>

Allows users to start multiple MPI applications via an option file. All applications must be registered with Legion and have a common `MPI_COMM_WORLD`.

-n <number of processors> Specify the number of processors on which the program will run.

Supported <flags> are⁶:

- v** Verbose option (up to four can be specified for increased detail).
- h** <host set context> Specify a set of hosts on which the program will run. The context path should not resolve to a single host but to a context containing the names of one or more hosts. You'll need to create it by hand. In the default setting, Legion will pick a compatible host and try to run the object. If this fails Legion will try another compatible host.
- HF** <local specification file> Use a specification file in your local file space to schedule objects from the current execution (see page 89 for more information).
- hf** <Legion specification file> Use a specification file in context space to schedule objects from the current execution (see page 89 for more information).
- in** <Legion input file> Tells Legion to copy the contents of a Legion file object to a file in the remote program's current working directory before execution. The remote file will have the same name as the Legion file. Note that this file will only be passed to node 0 unless you use -a.
 This flag can be used multiple times to indicate multiple input files. You can also use wild cards to specify groups of files.
- out** <output file> Name of an output file that should be copied from the remote program's current working directory into your context space after the program terminates.⁷ Note that this file will only be passed to node 0 unless you use the -a flag.
 The new file object will have the same name as the original output file object unless -a is used, in which case the name will be <name>.<mpi object

⁶ There is set of special flags for running in a fault tolerant mode listed on page 90.

⁷ The output files will be copied out when the program calls `MPI_Finalize()` or `MPI_Abort()`.

- id>. This flag can be used multiple times. You can also use wild cards to specify groups of files.
- Output files that are not found in the program's current working directory will be copied but will appear as empty files. If the program crashes, the files will be left in the program's current working directory.
- IN** <local input file> Tells Legion to copy a local file to the remote program's current working directory before execution. The remote file will have the same name as the local file. Note that this file will only be passed to node 0 unless you use -a.
- This flag can be used multiple times to indicate multiple input files. You can also use wild cards to specify groups of files.
- OUT** <output file> Name of an output file that should be copied from the remote program's current working directory into your local directory after the program terminates.⁸ Note that this file will only be passed to node 0 unless you use the -a flag.
- The new file will have the same name as the original output file unless -a is used, in which case the name will be <name>.<mpi object id>. This flag can be used multiple times. You can also use wild cards to specify groups of files.
- Output files that are not found in the program's current working directory will be copied but will appear as empty files. If the program crashes, the files will be left in the program's current working directory.
- a** Pass the files in the -in/-out/-IN/-OUT parameters to all nodes. The default setting passes these files only to node 0.
- Files in the -out/-OUT parameters will be copied to files that use the following conventions: <name>.<mpi object id>, where <name> is the <output file> specified in the -out/-OUT parameters.

⁸ The output files will be copied out when the program calls `MPI_Finalize()` or `MPI_Abort()`.

- stdin** <context path> Map standard input to a Legion file object. By default, this file object will only be passed to node 0 unless you include -A.
- stdout** <context path> Map standard output to a Legion file object. By default, this file object will only be passed to node 0 unless you include -A.
- stderr** <context path> Map standard error to a Legion file object. By default, this file object will only be passed to node 0 unless you include -A.
- STDIN** <local file> Map standard input to a local file. By default, this file will only be passed to node 0 unless you include -A.
- STDOUT** <local file> Map standard output to a local file. By default, this file will only be passed to node 0 unless you include -A.
- STDERR** <local file> Map standard error to a local file. By default, this file will only be passed to node 0 unless you include -A.
- A** Pass the files in the -std*/-STD* parameters to all nodes. The default setting gives these files only to node 0.
- p** <context path> Specify a context path for the program's instance LOIDs. If the context does not exist it will be created. The default path is:

/mpi/instances/<program_name>

If security is enabled, the default is:

/home/<user_name>/mpi/instances/<program_name>

- 0** <host context name> Run the first process (process zero) on this node.
- d** <Unix path name> Specify that all children change to the specified directory before they begin to run.
- S** Print statistics at exit.
- D** <variable_name>=<value>
Set the environment variable named in <variable_name> to a specified value on all MPI processes after they have called MPI_Init(). This option may be repeated multiple times to set additional environment variables.

-help Print command syntax and exit.

Assuming that you do not use an option file, the MPI application named in the `legion_mpi_run` command will be started with the given flags. For example, to run MPI program `vdelay` on two hosts you would enter:

```
$ legion_mpi_run -n 2 /mpi/programs/vdelay
```

Use `legion_ls` to examine the running objects of your application. The `/mpi/instances/<program_name>` context is the default location of your application's instances⁹:

```
$ legion_ls /mpi/instances/vdelay
```

If you run multiple versions of an application simultaneously, you can use the `-p` flag to specify a different context to hold your instances.

Specification file

A specification file is a text file that contains a list of hosts and the number of Legion objects that can be scheduled on those hosts. Objects from your application(s) will be scheduled on these hosts. You can use `legion_make_schedule` to produce this file or create it by hand. List one host and one integer indicating the number of objects the host can create (the default is 1) per line. List hosts by their Legion context names. A host can be listed multiple times in one file, in which case the integer values accumulate. E.g.:

```
/hosts/BootstrapHost      5
/hosts/slowMachine1
/hosts/bigPBSqueue        100
/hosts/slowMachine2       1
```

The file can be in your local directory or context space. Use `-hf` or `-HF` to use the file when running `legion_mpi_run`. All of the objects from this Legion MPI run will be created using vector creates.

Wild cards

You can also use wild cards to work with groups of input/output files. The following wild cards can be used with `-in/-out` and `-IN/-OUT`:

- *** Match 0 or more characters.
- ?** Match any one character.
- [-]** Match any character listed between the brackets (use these to specify a range of characters).
- ** Treat the character as a literal.

⁹ In a secure system, the default context is `/home/<user_name>/mpi/instances/<program_name>`.

For example, if you wanted to identify `done . 1`, `done . 2`, `done . 3` ... `done . 9` as your inputs, you could use square brackets to identify them as a group:

```
$ legion_mpi_run -n 2 -IN done.[0-9] \
  /mpi/programs/mpiFoo
```

You can use wild cards on the command line or in an option file. They can only be used with file names, however, not with directories.

Option file

The option file contains a list of all MPI applications that you wish to execute as well as any necessary arguments, as they would appear on the command line. All applications must have been registered (with `legion_mpi_register`) and use a common `MPI_COMM_WORLD`. The file must list one binary (and any arguments) per line. Each line can also contain one or more `legion_mpi_run` flags (but not `-f`). For example:

```
-n 2 /mpi/programs/mpitest
-n 3 /mpi/programs/mpitest_c
```

This would start a run of five instances (two of `mpitest` and three of `mpitest_c`).

If you use an option file, Legion will ignore the `-n` flag, program name, and any arguments given on the command line. Any `<flags>` will be treated as defaults and applied to all processes executed by this command (unless otherwise specified in the option file).

Fault tolerant mode

There is a set of special `legion_mpi_run` flags for running in a fault tolerant mode.

- ft** Turn on fault tolerance.
- s <checkpoint server>** Specifies the checkpoint server to use.
- R <checkpoint server>** Recovery mode.
- g <ping interval>** Ping interval. Default is 90 seconds.
- r <reconfiguration interval>** When failure is detected (an object has not responded in the last `x` seconds) restart the application from the last consistent checkpoint. Default value is 360 seconds.

These flags are used in specific combinations. You must use the `-ft` in all cases and you must use *either* `-s` or `-R`. The `-g` and `-r` flags are optional. Please see section 10.1.10 on page 61 in the Basic User Manual for more information on running in fault tolerant mode.

```
legion_native_mpi_config_host
  [<wrapper>]
  [-debug] [-help]
```

This command sets native MPI properties on a host. The following options are supported:

<wrapper>	Specify a wrapper script that locates <code>mpirun</code> on the host. The default specifies the <code>legion_native_mpich_wrapper</code> script, which is for an MPICH implementation.
-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

```
legion_native_mpi_init
  [<architecture>]
  [-debug] [-help]
```

This command installs the `legion_native_mpi_backend` class in a native MPI host.

The following options are supported:

<architecture>	Specify an architecture for which an implementation for this class can be registered.
-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

```
legion_native_mpi_register
  <class name> <binary path> <architecture>
  [-help]
```

Register a native MPI program with your Legion system. The example below registers `/myMPIprograms/charmm` (the binary path) as using a Linux architecture.

```
$ legion_native_mpi_register charmm \
  /myMPIprograms/charmm linux
```

You can run register a program multiple times, perhaps with different architectures or different platforms. If you have not registered this program before, this will create a context in the current context path (the context's name will be the program's class name – in this case, `charmm`) and registers the name in Legion context space.

The following option is supported:

-help	Print command syntax and exit.
--------------	--------------------------------

legion_native_mpi_run

```
[-v] [-a <architecture>] [-h <host context path>]
[-IN <local input file>] [-OUT <local result file>]
[-in <Legion input file>] [-out <Legion result file>]
[-n <nodes>] [-t <minutes>] [-legion]
[-help] [-debug]
<program context path> [<arg 1> <arg 2> ... <arg n>]
```

Start a native MPI program.

The following parameters are used for this command:

-h <host context path>	Specify the host on which the program will run. The default setting is the system's default placement, which is to pick a compatible host and try to run the object. If the host fails, the system will try another compatible host.
-v	Verbose option.
-a <architecture>	Specify the architecture to run on.
-IN <local input file>	Specify an input file that should be copied to the remote run from the local file system.
-OUT <local result file>	Specify a result file that should be copied back from the remote run to the local file system.
-in <Legion input file>	Specify an input file that should be copied to the remote run from the Legion context space.
-out <Legion result file>	Specify a result file that should be copied out from the remote run into Legion context space.
-n <nodes>	Specify the number of nodes for the remote MPI run.
-t <minutes>	Specify the amount of time requested for the remote MPI run. This option is only meaningful if the host selected to run the remote program enforces time limits for jobs.
-legion	Indicate whether the application makes Legion calls (see section 10.2.5 on page 68 in the Basic User manual).
-help	Print command syntax and exit.
-debug	Catch and print Legion exceptions.
<arg1> <arg2> ... <argn>	Arguments to be passed to the remote MPI program.

legion_nq

```

[-help] [-w] [-v]
[-a <arch> ] [-h <host context path>]
[-in <context name>] [-out <context name>]
[-IN <local file name> ] [-OUT <local file name>]
[-f <options file> ] [-novrun] [-t <minutes>]
[-n <nodes>] [-r <minutes>] [-p <priority>] [-d]
<program class> [ <arg1> <arg2> ... <argn>]

```

The `legion_nq` command submits jobs to a Legion JobQueue object. When a job is submitted, the JobQueue stores all necessary information for handling this job and returns a “ticket” which can be used for future references to the job. This command can be used in place of `legion_run` (page 100) to start a job on a remote Legion host. You cannot use `legion_probe_run` (page 96) to probe a `legion_nq` job. You can use `legion_manage_job` (page 82) to manage the job after it has started.

You can use the `-in/-out` and `-IN/-OUT` options to copy input and output files between your local host, the remote host, and context space. You can also use an options file.

Use `-p` to set a job's priority in the priority queue. Priorities are used when the number of submitted jobs is larger than the number of jobs that are allowed to execute simultaneously. If so, the job with the highest priority will be allowed to execute when a slot becomes available.

The command will propagate tty information to the JobQueue object so that stdout and stderr output can be redirected to `legion_nq` window

JobQueue

A JobQueue can be used to start, monitor, and clean up remote jobs. The JobQueue object is owned by the Legion system administrator (i.e., the `admin` user) and takes care of preparing, starting, and monitoring the job's progress. It assigns a ticket to each submitted job and a job record, which hold job-related information, and puts the job in a priority queue. The system administrator sets a maximum number of simultaneously running jobs (with `legion_manage_queue`, page 83) and the JobQueue tries to start jobs as slots become available.

If the job starts successfully, its record is moved to the list of running jobs. If it fails, its record is moved to the list of failed jobs. The job will be retried after another job has been started or has failed or the number of job slots is increased. Waiting jobs can be given higher and lower levels of priority by the job owners and/or the system administrator.

The JobQueue does not block and wait for a job to finish. When a job terminates (for whatever reason), its job proxy object (an object on the remote host which actually starts the job) notifies the JobQueue object.

The JobQueue also periodically pings running jobs, updates job statistics, kills jobs that have either overrun their maximum allocated running time, and removes jobs records marked for deletion.

A Legion system can have multiple JobQueue objects, responsible for different parts of the system's resources. You can use `legion_manage_queue` to manage individual JobQueue objects.

Note that jobs run with the JobQueue still must be registered, with either the `legion_register_program` (page 99) or `legion_register_runnable` (page 99).

Parameter

The following parameter is used with this command:

<program class> Specifies the program class of which an instance should be executed. This program class should have been previously created with the `legion_register_program` (page 99) or `legion_register_runnable` (page 99) command.

Options

The following options are supported

-help	Print this help message.
-w	Display standard I/O from the remote program.
-v	Verbose mode.
-a <architecture>	Specify the preferred Legion platform type on which the remote program should run.
-h <host context path>	Specify the context path of the Legion host object on which the remote program should run.
-in <context path>	Copy the specified Legion file to the current working directory of the remote program.
-out <context path>	Copy the specified Legion file's contents out of the current working directory of the remote program into Legion context space. The remote object's path should be the same as the <context path>.
-IN <local file name>	Send the specified local file to the current working directory of the remote program.
-OUT <local file name>	Copy a file out of the current working directory of the remote program into the specified path in your local file system.
-f <options file>	Obtain options from the specified file. An options file can contain one or more <code>legion_nq</code> flags delimited by spaces,

	tabs, or newlines. An options file is especially useful if your program uses several <code>-in/-out</code> or <code>-IN/-OUT</code> flags. You can include any flags but not arbitrary command-line arguments.
-novrun	Disable use of the virtual architecture on virtual hosts. The remote program will run on the physical architecture for the selected host. This option is not usually appropriate for user tasks. WARNING: the registered implementation for the virtual architecture of the selected host should also be usable on the physical architecture of the selected host if this option is used (e.g., a script).
-t <minutes>	Specify a requested amount of time (in minutes). This option is only meaningful if the host selected to run the remote program enforces time limits for jobs. Otherwise, this option is not required.
-n <nodes>	If the remote program is a native parallel job (e.g., a program written to use the local vendor MPI implementation), this option selects the number of nodes that should be allocated for the job. This option is not meaningful if the host selected to run the remote program is a uniprocessor, or does not support multi-node allocations.
-r <minutes>	Specify the maximum amount of time (in minutes) that the queuing system will wait for the program to complete before attempting to restart it. Default is 0 (no restarts).
-p <priority>	Specify job priority for the queuing system. Value range from 0 to 9, 0 being the highest priority. The default is 5. Users can assign values from 5 to 9. Only system administrators can assign values 0 to 4 (after job submission) with the <code>legion_manage_job</code> (page 82) command.
-q <context path>	Specify a JobQueue object. Default is <code>/etc/JobQueue</code> .
-d	Deactivate <code>-IN</code> files and temporary contexts to save process slots.
<arg1> <arg2> ... <argn>	Provide arbitrary command-line arguments for the program.

legion_probe_run

```

[-help] [-debug] [-v[erbose]]
[-in <context path>] [-out <context path>]
[-IN <local file path>] [-OUT <local file path>]
[-showscratch] [-setscratch <context path>]
[-pwd] [-hostname] [-list]
[-stat <remote file path>] [-delete <remote file name>]
[-statjob] [-signal <number>] [-kill]
{-l <probe LOID> | [-p[robe]] <local file path>}

```

This command checks a program started on a remote host with `legion_run` (page 100). It can also pass input files to or pick up output files from the remote host, check individual files associated with the job, and clean up the remote host space. It can be used with blocking or nonblocking runs, although it is especially useful with nonblocking runs.

You must have used the `-p` option when starting `legion_run` in order to use this command: this option creates a local file called a *probe file* that can contact your remote job. If you do not have a probe file for your job, `legion_probe_run` will be unable to locate your job.

Note that if you start a nonblocking run, the remote host will hold the results for six hours after the job terminates. If you do nothing, the host will tar up the job's current working directory and move it into your context scratch space. If you run `legion_probe_run` at any point (whether to retrieve output files or just to check the job's status) during those six hours, the clock restarts and the remote host will wait for another six hours before tarring and moving the job's directory. If you do not use the `-kill` option to clean up the job's directory, it will eventually be tarred and moved.

One of the following parameters must be used:

- p[robe]** <local file name> The name of the local probe file associated with the desired job.
- l** <probe LOID> The probe's LOID (may not be supported in all releases.)

The following options are supported. Options may be repeated. The order of specified options (except `-v[erbose]`, `-debug`, `-help`, and `-p[robe]`) determines order in which tasks will be done.

- debug** Catch and print Legion exceptions.
- help** Print command syntax and exit.
- v[erbose]** Run command in verbose mode.
- in** <context path name> Context path of a Legion file object whose contents should be copied into the remote program's current working directory.
The local file will have the same name as the <context path name> basename.

- out** <context path name> Context path of a Legion file object whose contents should be copied out of the remote program's current working directory. If the file doesn't exist, you will get an error message. The local source file will have the same name as the <context path name> basename. Output files that are not found in the program's current working directory are skipped.
- IN** <local file name> Similar to the -in option, but operates on a file in legion_probe_run's local execution environment (i.e., the file named in <local file name>).
- OUT** <local file name> Similar to the -out option, but operates on a file in legion_probe_run's local execution environment (i.e., the file named in <local file name>).
- setscratch** <context path> Specify which part of your context space should be used as scratch space for this remote job. Default is /tmp.
- showscratch** Print the currently set context scratch space. This will be the most recently specified space.
- pwd** Print the remote job's present working directory. This is useful if you set this directory with the -d option when you ran legion_run.
- hostname** Print the remote host's DNS name.
- list** Print a list of files in the remote job's current working directory.
- stat** <remote file name> Print the status of a particular file in the remote job's current working directory.
- delete** <remote file name> Delete the specified file from the remote working directory.
- statjob** Print a summary of the remote job's status. Possible values are "Running", "Error", and "Done".
- signal** <number> Send a signal to remote job (see kill -l).
- kill** Kill the remote job (if it hasn't yet terminated) and clean up its remote

current working directory. Please note that if you have not yet retrieved output files, they will be lost.

If you started `legion_run` in nonblocking mode and you do not clean up the remote directory after the job finishes, the remote host will wait six hours and then tar and move the entire directory to your context scratch space.

If you started `legion_run` in blocking mode and you then kill the remote job, `legion_run` will hang and must be killed by hand.

`legion_pvm_register`

`<class path name> <binary local path name> <platform type>`
`[-help]`

Registers a PVM task implementation. This setup is not necessary for tasks that will only be started from the command line (tasks that will not be "spawned"). Given the class named in `<class path name>`, the binary named in `<binary local path register>`, and the architecture type (currently Linux, Solaris, or SGI), this command creates an implementation object that can then be used by the Legion PVM Library.

Once you've registered an application, you can run it. You do not need a special command to run a registered PVM program. If necessary, you can examine Legion context space with either

```
$ legion_ls /pvm
```

to list the Tids of running PVM tasks, or

```
$ legion_ls /pvm/tasks
```

to list the registered task classes. You can also use Legion class utilities to examine the class state (e.g., `legion_list_instances` [page 17]).

For example, to register a Linux binary named `matmult` in Legion, enter:

```
$ legion_pvm_register /pvm/tasks/matmult matmult linux
```

The following option is supported:

-help Print command syntax and exit.

legion_register_program

```
<program class> <executable path> <legion arch>
[-debug] [-help]
```

This command allows users to register an independent (i.e., not linked to the Legion libraries) executable program (specified by the `<executable path>` argument) and make the program available for use within the Legion system. If the program was not previously registered, a new class object the registered program will be associated with a Legion class object, named in `<program class>`, the class and the context path will be created. The registered program will execute only on hosts of the architecture specified in `<legion arch>`.

Programs registered through `legion_register_program` can be executed with the `legion_run` command (page 100). See also `legion_register_runnable` for information about registering Legion programs (below).

The following parameters are used with this command:

<code><program class></code>	The Legion context path name of the class with which the registered program should be associated.
<code><executable path></code>	The local file path of the executable program to register. This can be any program that could be run from the shell command prompt, including scripts, and binary executable generated by any programming language.
<code><legion arch></code>	The platform type on which the program should be executed.

The following options are supported:

-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

For more information please see "Independent programs" on page 36 in the Basic User Manual.

legion_register_runnable

```
<program class> <executable path> <legion arch>
[-debug] [-help]
```

The `legion_register_runnable` command is similar to the `legion_register_program` command in that it allows programs to be registered for execution through the `legion_run` utility. However, whereas the `legion_register_program` tool is used to register independent programs, `legion_register_runnable` is used to register Legion-linked programs: programs that are linked against the Legion libraries and export the "runnable" object interface.

The following parameters are used with this command:

<code><program class></code>	The Legion context space path of the class with which the registered program should be associated.
------------------------------------	--

<executable path> The local file path of the executable program to register. This program should be a valid Legion object implementation that has been linked with the Legion library, and that exports the Legion "runnable" interface.

<legion arch> The platform type on which the program should be executed.

The following options are supported:

-debug Catch and print Legion exceptions.
-help Print command syntax and exit.

For more information please see "Legion-linked programs" on page 37 in the Basic User Manual.

legion_run

```
[ -help ] [ -debug ] [ -v[erbose] ] [ -w ] [ -a <arch> ] [ -h <host> ]
[ -block ] [ -non[-]block ] [ -d[ir] <dir> ] [ -D <var=value> ]
[ -in <contextfile> ] [ -out <contextfile> ]
[ -IN <localfile> ] [ -OUT <localfile> ]
[ -stdin <localfile> ] [ -stdout <localfile> ] [ -stderr <localfile> ]
[ -novrun ] [ -p[robe] <localfile> ] [ -setscratch <context> ]
[ -meta <option=value> ]
[ -f <options file> ] <program class> [ <arg1> ... <argn> ]
```

The `legion_run` command executes a single instance of a program associated with the program class specified in the `<program class>` argument. Legion will randomly select a host to execute the program (observing the restriction that only hosts with an acceptable architecture may be selected). Arbitrary command-line arguments may be specified for the remote program.

Any number of input and output files may be specified for a single execution of `legion_run` (i.e., the `-in/-out` and `-IN/-OUT` options can be repeated).

Please note that the `-t` and `-n` flags are deprecated as of version 1.8. Instead, you can use `-meta nodes=<num>` for `-t` and `-meta minutes=<num>` for `-n`.

The following parameters are used with this command:

<program class> Specifies the program class of which an instance should be executed. This program class should have been previously created with `legion_register_program` (page 99) or `legion_register_runnable` (page 99).

The following optional parameters are supported:

-debug Catch and print Legion exceptions.
-help Print command syntax and exit.
-v[erbose] Run command in verbose mode.

- w** Specifies that the set tty object should be used. If no tty object has been set, the flag will be ignored. If you have not created and set a tty object for your current window, you will not be able to see the command's output and an error message will appear. Please see "About Legion tty objects" on page 75 in the Basic User Manual for more information.
- a <architecture>** Allows users to specify what kind of architecture the program should be executed on.
- h <host context path>** Specify a remote host to run the program.
- non[-]block** Specify that the legion_run command be nonblocking. That is, once it starts your program on the remote host, it will exit and free up your command line.¹⁰ This command's default setting is to block.
- block** Specify that the legion_run command block. This is the default setting: when you run it, the command will continue to run at the command line until the remote job is finished.
- d[ir] <remote host directory>** Specify that the program run on a specified directory on the remote host.
- D <var=value>** Set environment variable <var> with <value> and run.
- in <context path name>** Copy the specified Legion file object into the remote program's current working directory before execution begins. The local file will have the same name as the <context path name> basename.
- out <context path name>** Copy the specified Legion file object out of the remote program's current working directory after the program terminates. The local source file will have the same name as Output files are copied out regardless of the cause of program termination, so partial results will be available if the program crashes. Output files not found in the

¹⁰ In all cases, legion_run blocks for ten seconds after you start the command. This is to be sure that there are no fatal errors that prevent the job from starting, such as a dead remote host or conflicting parameters.

program's current working directory will be skipped.

-IN <local file name> Similar to the **-in** option, but operates on a file in the local execution environment of `legion_run` (i.e., the file named in <local file name>).

-OUT <local file name>

Similar to the **-out** option, but operates on a file in the local execution environment of `legion_run` (i.e., the file named in <local file name>).

-stdin <local file name>

Specify a local file from which the remote job gets its standard input.

-stdout <local file name>

Specify a local file to which the remote job writes its standard output.

-stderr <local file name>

Specify a local file to which the remote job writes its standard error.

-novrun

Do not run on a virtual host. I.e., disable use of the "virtual architecture" on virtual hosts. The remote program will run on the physical architecture for the selected host. This option is usually not appropriate for user tasks. Warning: the registered implementation for the virtual architecture of the selected host should be usable on the physical architecture of the selected host (e.g., a script).

-meta <option=value> Specify metaoptions for the run. These options may be meaningful only for certain combinations of jobs and machines. Listed below is the set of supported options and their meaning.

`nodes=<min>[, <max>]` Number of nodes for job (default is 1).

`tasks_per_node=<num>` Number of tasks per node (default is 1). Total number of tasks is product of allocated nodes and `tasks_per_node`.

`queue=<name>` Name of a queue on host.

`priority=<str>` Priority to run under.

`minutes=<num>` Duration of run.

-p[robe] <local file name>

Store the remote job's tracking name in a file on your local host. You must use this option if you wish to use the `legion_probe_run` tool.

-setscratch <context path>

Specify a portion of your context space to be used as scratch space.

-f <option file>

Allows users to specify options for running `legion_run` in a separate file rather than listing them on the command line. This is useful for programs that make extensive use of the `-in/-out` or `-IN/-OUT` options. The file can contain one or more of the `legion_run` flags delimited by spaces, tabs, or blank lines. The program class name and any arbitrary command-line arguments may not be included in this file. No other information should be included.

<arg1> <arg2> ... <argn>

Allows users to provide arbitrary command-line arguments for the program.

The possible architectures that can be used with the `-a` flag are limited. At the moment they are:

Architecture	Corresponds to	Comments
<code>solaris</code>	Sun workstations running Solaris 5.x	
<code>sgi[n32 n64]</code>	SGI workstations running IRIX 6.5	
<code>linux</code>	x86 running Red Hat 6.x Linux	
<code>alpha_linux</code>	DEC Alphas running Red Hat 6.x Linux	
<code>alpha_DEC</code>	DEC Alphas running OSF1 v4	
<code>rs6000</code>	IBM RS/6000s running AIX 4.2.1	
<code>hppa_hpux</code>	HPUX 11	
<code>t90</code>	Cray T90s running Unicos 10.x	virtual hosts only (see page 69 in the System Administrator Manual)
<code>t3e</code>	Cray T3E running Unicos/mk 2.x	virtual hosts only (see page 69 in the System Administrator Manual)

For more information on this command please see "Running a Legion application" on page 35 in the Basic User Manual.

legion_run_multi

```

[-help] [-debug] [-v[erbose]] [-z[ero]] [-r[estart]]
[-e[xec] command] [-x <exceptfile>] [-a[rch] <architecture>]
[-d[ir] <dir>] [-D <var=value>]
{-n <number of processors> | -s[chedule] <schedfile>}
-f[ile] <specfile> [-t[ime] <timefile>] [--] <program class>
[<arg1> <arg2> ... <argn>]

```

This command runs a previously registered serial program, named in <program class name>, with all of the different input files, using a simple specification file, named in <specification file name>, to describe the names of the expected input and output files. You must provide the number of processors and/or a schedule file.

Before you run this command, you must have registered the program name and created a specification file. The specification file format is:

keyword	filename	pattern
---------	----------	---------

You must fill in all three fields on each line. One file per line.

Specification file

The first field, the keyword, provides information about the file named in the second field. Possible keywords are IN/in, OUT/out, CONSTANT/constant, and stdin/stdout/stderr. The case determines the file's location (except for std*):

constant/in/out	Legion context space
CONSTANT/IN/OUT/	local file space
stdin/stdout/stderr	

The CONSTANT/constant keywords indicate identical input for multiple runs (e.g., input that is used for each run, such as a password). The std* keywords refer to local files that stdin, stdout, and stderr use. A separate stdout and stderr are created for each run. If you do not indicate a stdout file, standard output will be sent to your Legion tty object. Multiple IN/in, OUT/out, and CONSTANT/constant files may be specified, one per line.

The second field is the name of an input or output file that your program expects to read or write to in the local directory of whichever host it runs on.

The third field is the naming pattern for input and output files. You may use "*" as a pattern holder for IN/in and OUT/out files. The pattern holder allows you to specify large groups of files in different directories for input and output.

For example, suppose that you want to run program Foo. Foo needs two input files, FooFoo and MyFoo, and a password. It also produces one output file, Bar. Your specification file would look like this:

IN	FooFoo	/mylocaldir/foo*.ent
in	MyFoo	/mycontext/bar*.gif
OUT	Bar	/mylocaldir/output*.stuff
CONSTANT	secret	/etc/passwd

Legion looks in local space for files that fit the specified patterns and finds five sets of files:

IN LOCAL SPACE	IN CONTEXT SPACE
/mylocaldir/foo1.ent	/mycontext/bar1.gif
/mylocaldir/foo2.ent	/mycontext/bar2.gif
/mylocaldir/foo3.ent	/mycontext/bar3.gif
/mylocaldir/foo4.ent	/mycontext/barA.gif
/mylocaldir/foo5.ent	/mycontext/barB.gif

Notice that the first three files in each column have common patterns (1, 2, and 3) whereas the last two do not (4 and 5, A and B). The files with mismatched patterns are discarded, leaving three sets of input files, or three jobs that can be run:

IN LOCAL SPACE	IN CONTEXT SPACE
/mylocaldir/foo1.ent	/mycontext/bar1.gif
/mylocaldir/foo2.ent	/mycontext/bar2.gif
/mylocaldir/foo3.ent	/mycontext/bar3.gif

These jobs can be identified by their pattern: 1, 2, and 3. Before going any further, Legion checks for any previously created output files. If it finds a file called /mylocaldir/output1.stuff it knows that job 1 has already been run. The remaining jobs, 2 and 3, have not. Legion then copies the remaining four input files to the two hosts that will run the jobs:

/mylocaldir/foo2.ent copied to-->	HostA/FooFoo
/mycontext/bar2.gif	HostA/MyFoo
/mylocaldir/foo3.ent	HostB/FooFoo
/mycontext/bar3.gif	HostB/MyFoo

The jobs are run and two output files, each called Bar, are created on HostA and on HostB. HostA/Bar's contents are copied to /mylocaldir/output2.stuff, a file in the original host's local directory space, signalling that job 2 is finished. HostB/Bar's contents are copied to /mylocaldir/output3.stuff, signalling that job 3 is finished.

A more complex specification file might look like this:

IN	FILE1	/usr/localtmp/dump/pdb*.ent
in	NEXT2	/home/an4m/pictures/*.gif
OUT	THEN3	output*
out	FILE4	./dssp*log
CONSTANT	Crypt	/etc/passwd
constant	Salt-File	/home/admin/user-list

The program must be written so as to read/write the files named in the second field from its current directory. In this case, the registered program must read FILE1, NEXT2, Crypt, and Salt-File, and write THEN3 and FILE4. When `legion_run_multi` is called on the registered program with the above specification file, it looks for matching patterns:

```

/usr/local/tmp/dump/pdb1egs2.ent in your local directory
/home/an4m/pictures/1egs2.gif in your Legion context

```

These are sent as input files for `run 1egs2`, which generates two output files:

```

output1egs2          in your local directory
./dssp1egs2log       in your Legion context

```

Exception file

The exceptions file is an addendum to the specification file. It specifies additional switches for specific patterns. This allows you greater flexibility and control when running large batches of jobs. The information in the exceptions file is passed directly to `legion_run` or, if you use the `-e` flag, to whichever command you have specified (`legion_run_multi` command is essentially a script that calls multiple `legion_runs` or whichever command specified).

For example, suppose that the following jobs match the patterns in your specification file:

```

ABC
DEFGH
123
9999
GIRAFFE

```

The specification file already provides input/output/constant information for all of these jobs and command line flags provide additional instructions. However, you wish to fine-tune these instructions for just jobs DEFGH and 123. In the exceptions file, you can specify these two jobs and give your additional instructions:

```

DEFGH -h /hosts/special_host_name
123 -v -IN /tmp/myJob/ExtraStuff

```

The first line tells Legion that you wish to run job DEFGH on a specific host. The second line says that job 123 requires an additional input file and that you wish to run the job in a verbose form.

These instructions combine with any other specifications that were provided on the command line and/or in the specification file.

Limitations, caveats, warnings

1. Be sure that you use legitimate syntax in the exceptions file. Do not specify flags that do not exist for that particular command (e.g., there are no `-CONSTANT/-constant` flags in `legion_run`).
2. If you specify a host in the exceptions file and an architecture on the command line (with the `-a` flag), be sure that they match. If you specify conflicting architectures Legion will be unable to process that particular job (but it will run any other jobs).

Parameters

You must specify at least one of the following two parameters:

-n <number of processors> Specify the number of processes running at one time.

-s[chedule] <schedule file>

Specify a set of host context paths and how many processes are to be run on each host (e.g., `/hosts/Foo 3` to run up to three jobs on host object Foo). The format of the schedule file is as below. One host per line.

<code>/hosts/firsthost</code>	<code>2</code>
<code>/hosts/secondhost</code>	<code>1</code>
<code>/hosts/thirdhost</code>	<code>7</code>

You must include both of the following parameters:

-f[ile] <specification file path>

Local path name of specification file.

<program class name>

The program's Legion class name (created when the program was registered in Legion).

Options

The following options are available with this command:

-help Print command syntax and exit.

-debug Turn debugging mode on. In this mode, `legion_run_multi` prints copious output regarding which patterns must be run. This flag also turns on the verbose mode (`-v`).

-v[erbose] Provide a verbose output as the command is running.

-z[ero] Consider input and output files of zero size to be valid. If this option is used, a zero-sized input file can be used to start a job and a zero-sized

	output file can be used to signal a completed job.
-r[estart]	Restart incomplete jobs (will attempt five times max).
-e[xec] <command>	Run some other <command>. The default is legion_run.
-x <exceptions file path>	The exceptions file's Unix path.
-a[rch] <architecture>	Platform architecture for remote hosts.
-d[ir] <dir>	Remote directory to which each job must change before execution.
-D <var=value>	Environment variable to be set for each job.
-t[ime] <timefile>	Specify a file for recording the job history of the processes. This file will list jobs by their name and gives the time that each job was checked, when it was checked (measured in seconds since January 1, 1970), the status that was returned (e.g., Started, Error), and (if known) the host that it was run on. The file is in your local directory.
--	End switches for legion_run_multi.
<arg1> <arg2>... <argn>	Provide arbitrary command-line arguments for the registered program.

The possible architectures that can be used with the -a flag are limited. At the moment they are:

Architecture	Corresponds to	Comments
solaris	Sun workstations running Solaris 5.x	
sgi	SGI workstations running IRIX 6.5	
linux	x86 running Red Hat 6.x Linux	
alpha_linux	DEC Alphas running Red Hat 6.x Linux	
alpha_DEC	DEC Alphas running OSF1 v4	
rs6000	IBM RS/6000s running AIX 4.2.1	
hppa_hpux	HPUX 11	
t90	Cray T90s running Unicos 10.x	virtual hosts only (see page 69 in the System Administrator Manual)
t3e	Cray T3E running Unicos/mk 2.x	virtual hosts only (see page 69 in the System Administrator Manual)

rpc.lmountd

```
[{-f <exports file local path>} |
  {--exports-file=<exports file local path>}]
[{-d | --debug} <facility>]
[{-P | --port} <port number>]
[-F | --foreground] [-h | --help]
[-n | --allow-non-root] [-p | --promiscuous]
[-t | --no-spoof-trace] [-v | --version]
```

The Legion NFS mount daemon enables you to NFS-mount a Legion file system to your local machine.¹¹ Your local machine must have access to a working Legion net. Please note that you must also run `lnfsd` (below) in order to start an Legion NFS session. Currently, the Legion NFS mount daemon only works on Linux and Alpha Linux platforms. Please contact us at [<legion-help@cs.virginia.edu>](mailto:legion-help@cs.virginia.edu) if you need to run it on another architecture.

Legion context space is mounted via the same means used to mount a normal NFS file system (i.e., `mount`). When you NFS-mount context space, `lmountd` checks your MOUNT request against your local exports file (the list of exported file systems). If your machine is listed, `lmountd` creates and returns a file handle and adds an entry in `/etc/rmtab`.

To end the session, run `umount`. Upon receipt of your UMOUNT request, `lmountd` removes your client's entry from `rmtab`. The client kernel unmounts the file system.

Please note that a Legion NFS mount does not give you permission to view or use Legion objects: if your Legion system requires a Legion login you must run `legion_login` (page 68) as a separate step.

The `lmountd` should be run locally.

Please read the Security and Example sections of the `rpc.lnfsd` command (page 111).

Exports file

The exports file contains a list of what file systems are available for export and which machines are allowed to mount those file systems. The `lmountd` reads the root of context space as `/legion`, so it will be listed in the exports file in the following format:

```
/legion <host name>(rw)
```

The `(rw)` indicates that the named host is allowed to read and write requests to `/legion`.

The export file's default path is `/etc/exports` but you can specify another path with the `-f/--exports-file` flag.

¹¹ Legion NFS mount relies on the `mount` and `umount` Unix command-line utilities. Please see the associated man pages for information about using these tools. For more information about NFS mounts, you can also see the `exports`, `mountd`, and `lnfsd` man pages. The two Legion NFS daemons and supporting materials were derived from the Unix User-Space NFS Server Version 2.2. Key contributors/authors were Mark Shand, Don Becker, Rick Sladkey, and Olaf Kirch.

The following options are supported:

- f | --exports file** <exports file local path>
Specify a local path for the exports file (page 109). Default is /etc/exports. For security reasons, the Legion NFS daemons are run locally, so only the local host should be listed in this file. In the future, this will be mandatory and this option will be eliminated.
- d | --debug <facility>**
Log operations verbosely. The current legal values for <facility> are call for logging RPC calls and arguments, fhcache for file handle cache operation, and auth for authentication routines.
- P | --port <port number>**
Specify a port number for `lmountd`. Default is a random number under 1024. Please note that unlike NFS `mountd`, `lmountd` does not register with `portmap` (this ensures that NFS RPC requests bound for NFS servers or `mount` daemons are not inadvertently intercepted). So while you do not have to specify a port, this implies that you should.
- F | --foreground**
Run `lmountd` in the foreground. If debugging is also requested, it will be sent to `stderr`.
- h | --help**
Print command syntax and exit.
- n | --allow-non-root**
Allow incoming NFS-mount requests to be honored even if they do not originate from reserved IP ports. Please read the Security section of `rpc.Infsd` (page 111), before using this option.
- p | --promiscuous**
Run in promiscuous mode, so that the server will serve any host on the network. This violates `lnfsd`'s security premises, so it is deprecated.
- t | --no-spoof-trace**
By default, `lmountd` logs every access by unauthorized clients. This option turns off logging of such spoof attempts for all hosts listed explicitly in the exports file.
- v | --version**
Print the current version number of the program.

The daemon recognizes the following signals:

SIGHUP	Causes <code>lmountd</code> to reread the exports file and any access restrictions in <code>/etc/hosts.allow</code> and <code>/etc/hosts.deny</code> . If you have altered these files and wish your changes to take effect, you must send SIGHUP to <code>lnfsd</code> as well.
SIGTERM	Kills the daemon. This is the preferred method, since it allows proper termination of the object.
SIGUSR1	If <code>lmountd</code> is invoked with debugging options, this signal toggles generation of debugging information

`rpc.lnfsd`

```
[{-f <exports file local path>} |
  {--exports-file=<exports file local path>}]
[{-d | --debug} <facility>]
[{-P | --port} <port number>]
[-F | --foreground] [-h | --help]
[-l | --log-transfers] [-n | --allow-non-root]
[-p | --promiscuous] [-t | --no-spoof-trace]
[-v | --version]
```

The Legion NFS process server daemon interposes an NFS client and the Legion file system.¹² It is a user-level process derived from Linux `nfsd`. It accepts NFS RPC requests from a kernel client and translates them to invocations on Legion objects. The results are returned to the daemon, translated into an RPC reply, and forwarded to the client. Currently, the Legion NFS process server daemon only works on Linux and Alpha Linux platforms. Please contact us at `<legion-help@cs.virginia.edu>` if you need to run it on another architecture.

The `lnfsd` should be run locally.

You must also run `lmountd` (with `rpc.lmountd`, page 109) in order to mount Legion NFS. Both of these daemons must be on the client Legion machine. We recommend that you run them as root (see Security, below, for more information).

Security

The two Legion NFS daemons, `lnfsd` and `lmountd`, provide an NFS-like file system while preserving Legion's stronger security features. The NFS client authenticates itself to `lnfsd` by passing the requesting user's Unix uid. This authentication is predicated on trust between the client's kernel (and thus NFS) and the NFS daemon (in this case, `lnfsd`). The interface between the NFS kernel client and `lnfsd` follows NFS protocol.

¹² Legion NFS mount relies on the `mount` and `umount` Unix command-line utilities. Please see the associated man pages for information about using these tools. For more information about NFS mounts, you can also see the `exports`, `mountd`, and `nfsd` man pages. The two Legion NFS mount daemons and supporting materials were derived from the Unix User-Space NFS Server Version 2.2. Key contributors/authors were Mark Shand, Don Becker, Rick Sladkey, and Olaf Kirch.

We recommend that `lnfsd` run as root.

Both `lnfsd` and `lmountd` are responsible for bridging semantic differences between Unix and Legion security mechanisms and must translate a Unix uid to Legion credentials. Users must be logged in with the `legion_login` (page 68) utility to access objects in a secure Legion system. This utility creates a session file, corresponding to a Unix uid, in users' local `/tmp` directory. An object's ACL list determines who has access rights to the object. In an NFS-mounted Legion system, once a Unix uid has been mapped to Legion credentials access to Legion objects is authenticated and authorized via Legion mechanisms. This prevents unauthorized access to the object through Unix.

We expect that the daemons will execute with root privileges on the client's host and will accept access only from privileged ports. Assuming that root and root-privileged processes (including the NFS kernel client) are not compromised, `lnfsd` and `lmountd` are secure from malicious attacks. File handles are never exposed on the wire.

If a malicious remote user becomes root on the client host and impersonates a valid user by using a valid uid on an NFS request, he may attempt to subvert `lnfsd` via a bogus RPC request. These type of attacks are avoided by cryptographically signing file handles.

A malicious user may also attempt to subvert `lmountd`. However, the daemon will only respond to the local host, so a rogue user cannot obtain the file handle needed for subsequent transactions.

Legion uses fine-grained ACLs for authorization: they record individual and group users' abilities to invoke individual methods. Translations between Legion ACLs and Unix permission bits are therefore lossy. Individual methods must be classified as `read`, `write`, `execute`, or some combination of these in order to correspond to Unix permission bits.

Legion provides a global name space, including individual accounts, which may not correspond to any particular Unix account on a local machine. Legion does not assign group ownership to an object, so object ownership and groups within a Legion NFS mounted file system do not directly adhere to Unix semantics. A Legion object's owner may not hold an account on the client machine, so the owner's Legion credentials cannot be mapped to a Unix uid. Therefore, all objects in a Legion NFS mounted file system *appear* to be owned by the user accessing the objects. Unix permissions bits are set according to that user's ability to access the object. Group permissions mirror other permissions.

Example

To NFS-mount a Legion system, you need to start the two Legion NFS daemons, `lmountd` and `lnfsd`, then run `mount`. Both `lmountd` and `lnfsd` need static ports, which are passed to `mount`. We suggest that you use the `-P` option to manually select a port. These steps might look something like this:

```
$ rpc.lmound -f /local_path/exports -P 2001
$ rpc.lnfsd -f /local_path/exports -P 2000
$ mount -t nfs -o port=2000,mountport=2001 \
  MyHost:/legion /legion-dir
```

The first command starts `lmound` and the second `lnfsd`. The third mounts Legion in local file space. Note that the first two commands provide a path for the exports file and a port number. If you do not manually select a port one will be chosen, but you'll need to know the port number and pass it to `mount`.

Note that `lmound` interprets the root of Legion context space (/) as `/legion`. In the `mount` command, therefore, you must pass it `/legion` as the root of the directory to be mounted. In the example above, `/legion-dir` directory is the local path pointing to the mounted directory but you can use any existing path.

The following options are supported:

- f | --exports-file <exports file local path>**
Specify a local path for the exports file. Default is `/etc/exports`. For security reasons, the Legion NFS daemons are run locally, so only the local host should be listed in this file. In the future, this will be mandatory and this option will be eliminated.
- d | --debug <facility>**
Log operations verbosely. The current legal values for `<facility>` are `call` for logging RPC calls and arguments, `fhcache` for file handle cache operation, and `auth` for authentication routines.
- P | --port <port number>**
Specify a port number for `lnfsd`. The default is 2049. Note that unlike a normal NFS server, `lnfsd` does not register with `portmap` (this ensures that NFS RPC requests bound for NFS servers or mount daemons are not inadvertently intercepted).
- F | --foreground**
Run in the foreground. If debugging is also requested, it will be sent to `stderr`.
- h | --help**
Print command syntax and exit.
- l | --log-transfers**
Try to catch all files retried from and written to `lnfsd`. For each file stored or retrieved, a single line containing the client's IP address and the file's name is written to the system log daemon.

-n --allow-non-root	Allow incoming NFS requests to be honored even if they do not originate from reserved IP ports. Please see the Security section on page 111 before using this option.
-p --promiscuous	Run in promiscuous mode, so that the server will serve any host on the network. This violates <code>lnfsd</code> 's security premises, so it is deprecated.
-t --no-spoof-trace	By default, <code>lnfsd</code> logs every access by unauthorized clients. This option turns off logging of such spoof attempts for all hosts listed explicitly in the exports file.
-v --version	Print the current version number of the program.

The following signals are supported:

SIGHUP	Reread the exports file and flush the file handle cache.
SIGIOT	When compiled with the <code>-DCALL_PROFILING</code> option, this signal will cause <code>lnfsd</code> to dump the average execution times per NFS operation into <code>/tmp/nfsd.profile</code> .
SIGTERM	Kills the daemon. This is the preferred method, since it allows proper termination of the object.
SIGUSR1	If <code>lnfsd</code> is invoked with debugging options, this signal toggles generation of debugging information.

Known bugs

Currently, `chown` is unsupported, since the mapping between a Unix target UID and the corresponding Legion principal may not be available.

Legion contexts do not share the “append-only” implementations of Unix directories. On some platforms (such as x86 Linux), this discrepancy manifests itself as an error when you attempting to remove large directories that require multiple invocations of `readdir`. For example, if you run “`rm -rf`” on a large context some subcontext entries may be left intact. You will see an error message indicating that the parent context could not be deleted because it was not empty. To get around this, keep retrying the operation until all entries are removed and the parent context can be deleted.

3.0 Error codes

The `LegionClientExit.h` file contains the declarations for the error reporting mechanism for these command-line utilities:

```
legion_activate_instances
legion_activate_object
legion_cat
legion_classof
legion_context_add
legion_cp
legion_deactivate_instances
legion_deactivate_object
legion_get_host
legion_get_vault
legion_list_instances
legion_list_names
legion_list_objects
legion_login
legion_logout
legion_ls
legion_ping
legion_print_config
legion_pwd
legion_rm
legion_set_vault
legion_wellknown_class
legion_whereis
legion_whoami
```

Below are the identifiers and their codes.

```
LC_SUCCESS = 0
LC_BAD_ARGUMENTS = 1
LC_MEMORY_ALLOCATION_ERROR = 2
LC_OBJECT_NOT_FOUND = 3
LC_CONTEXT_NOT_FOUND = 4
LC_OBJECT_DOES_NOT_EXIST = 5
LC_OBJECT_ALREADY_EXISTS = 6
LC_LOID_NOT_OBTAINED = 7
LC_LEGION_OUTCALL_FAILED = 8
LC_SECURITY_EXCEPTION = 9
```

LC_GENERAL_LEGION_ERROR = 10
LC_UNKNOWN_LEGION_ERROR = 11
LC_OTHER_ERROR = 12

4.0 Internal tools

These tools are used by Legion objects and are not intended to be run from the command line.

`legion_accounting_info_pusher`

Used by the host object to “push” accounting information back to the `legion_get_accounting_info` tool (page 57). It is only used by the host object.

`legion_init_Apps_security`

Start the Application package's security. This command is automatically run when you run `legion_init_Apps` (page 67).

`legion_init_Extra_security`

Start the Extra package's security. This command is automatically run when you run `legion_init_Extra` (page 67).

`legion_init_HPC_security`

Start the HPC package's security. This command is automatically run when you run `legion_init_HPC` (page 67).

`legion_init_SDK`

Initiate the SDK package. This command is automatically run when you run `legion_initialize` (page 39).

`legion_init_SDK_security`

Start the SDK package's security. This command is automatically run when you run `legion_init_security` (page 67).

`legion_make_backend`

Backend script for `legion_make` (page 73). The Legion admin installs this tool as a class, usually in context space in `/class`, `/class/HPC`, or `/class/SDK`. Installed using `legion_register_program`.

`legion_modify_scheduler`

Used to configure any instance of the round-robin scheduler. Used to set the hosts from which the scheduler will select targets, the collection from which it gets its info, and so on.

`legion_mplfront`

This tool is the actual binary for the Legion MPLC compiler and is called by the `legion_mplc` script (page 75). It is responsible for parsing an MPL code and generating the appropriate C++ code.

legion_native_mpi_backend

Backend script for `legion_native_mpi_run`. The Legion admin installs this tool as a class, usually in context space in `/class` or `/class/HPC`. Installed using `legion_register_program`.

`legion_native_mpich_wrapper`
`legion_native_mpiibm_wrapper`
`legion_native_mpsgi_wrapper`
`legion_native_mpiultra_wrapper`

Site-specific scripts used to determine how to start native MPI jobs at that site. The Legion admin installs the appropriate tool when he or she runs `legion_native_mpi_config_host` for the host object running at that site.

legion_permit_user

This tool tells whether or not a specific Legion user can run on a specific host. If a blank string is returned, then the Legion user cannot run on that machine. If a non-blank string is returned the string represents a Unix/Windows user name, and the legion user runs as that Unix/Windows user on that machine.

Not necessarily an internal tool, because any user can run it. Specifically used by `legion_make_schedule` to determine whether or not a user can run on a chosen host.

legion_probe_object

`{-c <object context path> | -l <object LOID>}`

A debugging tool for contacting a specified object. Used for internal debugging. If the tool cannot contact the object, it tries to contact the relevant host and class hierarchy to find the cause of the problem.

legion_translate_arch

Internal tool used to translate architecture names (e.g., `linux`, `sgi`, `alpha_linux`, `solaris`, etc.) to Legion's internal numbers for those architectures (e.g., 1, 6, 4, 2, etc.) and vice versa.

`legion_vrun_run`
`legion_vrun_status`
`legion_vrun_kill`

These three scripts are used for virtual hosts: they sit on the physical host and are called by the virtual host object. Please see section 14.0 in the System Administrator Manual.

legion_watch_process

This is a wrapper used to manage a long-running process. If the process it is managing dies with a non-zero exit status, or is killed, the wrapper will automatically restart its child. If the child exits with an exit status of zero, the wrapper terminates. If the wrapper receives a `SIGINT` or a `SIGHUP`, it passes the signal along to the child and exits.

Manipulating hosts and vaults

5.0 Alternative ways to start and shut down

5.1 Starting extra hosts and vaults, command line



Please note that the procedure described below does not use `rsh` classes, as does the normal procedure described in “Adding a new host” and “Adding a new vault” (starting on page 52 in the System Administrator Manual). The `rsh` procedure¹³ is preferable, if possible, to the command-line procedure, since it is easier and faster.

The full procedure explained below can be used for adding new host or vault objects with the command-line procedure to a system spanning multiple file systems or to a Legion system with a common file system. For a system spanning multiple file systems, however, skip steps 2 and 3. In either case, the main Legion system must be active. Start it with the first-time start-up or restart procedure, if necessary.

1. Login to the desired machine.
2. Create the `$LEGION` and `$LEGION_OPR` directories on the new machine, if they do not already exist.
3. Copy the following files from your Legion system's `$LEGION_OPR` directory to the new machine's `$LEGION_OPR` directory, if they do not already exist.

```
legion_context_env.sh
legion_context_env.csh
LegionClass.config
LegionClass.config._LegionStorage_MetaData_
```

4. Set the environment on the new machine.

ksh or sh users:

```
. $LEGION/bin/legion_env.sh
. $LEGION_OPR/legion_context_env.sh
```

csh users:

```
source $LEGION/bin/legion_env.csh
source $LEGION_OPR/legion_context_env.csh
```

5. On the new machine, start up the host:

```
legion_startup -local
```

You will prompted as to whether or not you wish to start a host and vault object. We recommend that you answer "yes" to both.

¹³ If preferred, `ssh` can be used in place of `rsh`

5.2 Shutting down extra hosts and vaults, command line

1. Shut down the main system (see page 26 in the System Administrator Manual).
2. Login to the desired machine (i.e., the machine which contains the host and/or vault object that you wish to shut down)
3. Set the user environment correctly (see page 14 in the System Administrator Manual).
4. Enter the `legion_shutdown` command with the `-local` flag:

```
$ legion_shutdown -local
```

The vault or host object will then shut down.

5.3 Restarting extra hosts and vaults

Once an extra host or vault has been safely shut down, it can be restarted. If the host or vault was not safely shut down, you cannot restart it, but must delete the OPR directory, restart Legion, and start the host or vault again (see "System restart," pg. 27, in the System Administrator Manual).

1. Login to the machine on which the host or vault object resides.
2. Set the user environment (please see page 14 in the System Administrator Manual).
3. Run the start-up command with the `-local` flag:

```
legion_startup -local
```

Your local host or vault object now be running.

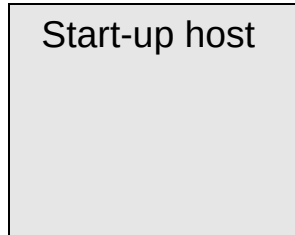
5.4 Using two hosts to start up a system

Generally, you can use a single host to start a new Legion system, but if your machines are strapped for memory you may need to split the process between two hosts. Technically, you can use two hosts used for the whole procedure, the start-up host and the bootstrap host (Figure 1, below): start Legion from one host and use a different host as the bootstrap.

The start-up host must always be used to run `legion_startup`. The host that you select during `legion_setup_state` and that contains the basic Legion classes is the start-up host. The bootstrap host is used to boot the system and is the first host object added to the system. It will contain the first pieces of the new system (the first contexts and instances of the classes). In either case, Legion binary executable files must be installed on any hosts used for Legion

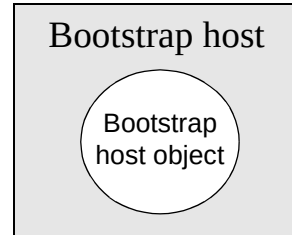
procedures. That is, you don't have to run `legion_startup` in order to use the host in a Legion net: you just have to have the binary files. You do not have to run `legion_initialize` on the same host on which you run `legion_setup_state` and `legion_startup`.

Figure 1: Start-up and Bootstrap hosts



Start-up host characteristics

- Contains Legion binary executable files
- No start-up host object



Bootstrap host characteristics

- Contains Legion binary executable files
- First host object added to system
- Contains first pieces of the system

Runtime library examples and interfaces

6.0 Example translations: the simple class

The simple class in this example is a nonsense class that does no meaningful computation. However, it exercises the library's full functionality and illustrates how our compiler generates translations. The example shows the C++ class, its translation, and a sample program that illustrates how we invoke methods.

To write the translation we first define a wrapper class called `Legion_Simple`. This wrapper contains a protected data member, an object of the translated C++ class. We call this the *wrapped object*. For each wrapped object member function we define a corresponding member function in `Legion_Simple`. The `Legion_Simple` member functions' job is to take a work unit, build a corresponding call to the wrapped object's member function, execute the call, and package up any return values.

`Legion_Simple` has additional member functions that figure out which wrapper member function should be called (`invoke_method()`), indicate to the invocation store which member functions will be accepted (`enable_functions()`), and perform the server loop (`accept_member_functions()`).

6.1 Simple.h and Simple.c

```
%-----Simple.h-----%
// A very simple class definition
#ifndef _H_Simple_
#define _H_Simple_

#include <stdio.h>

class Simple
{
    int data;
public:
    Simple();
    ~Simple();
    int op1(int foo);
    int op2(int &foo, int &bar);
};

#endif
%-----Simple.c-----%
// A very simple class definition
#ifndef _C_Simple_
#define _C_Simple_
```

```

#include <stdio.h>
#include "Simple.h"

Simple::Simple() {
    data = 0;
}

Simple::op1(int foo) {
    data = foo;
}

int
Simple::op2(int &foo, int &bar) {
    foo = data*data;
    bar = data+data;
    return foo+bar;
}

Simple::
~Simple()
{
}
#endif

```

6.2 Simple.trans.h and Simple.trans.c

```

%-----Simple.trans.h-----%
// Legion 'wrapper' class definition for the Simple
// class
#ifdef _H_Simple_trans_
#include <stdio.h>
#include "legion/Legion.h"
#include "Simple.h"
#define SIMPLE_OBJECT_CLASS_ID "Simple"
#define SIMPLE_OP1_FUNCTION_NUMBER 1
#define SIMPLE_OP2_FUNCTION_NUMBER 2

// class Legion_Simple
class Legion_Simple
{
private:
    // the object being wrapped
    Simple *object;

    // For generating methodDone events
    virtual void generate_MethodDoneEvent();

public:
    Legion_Simple();
    ~Legion_Simple();

    // wrapper member functions for each member
    // function in 'object'
    void Legion_op1(UVaL_Reference<LegionWorkUnit> wu);
    void Legion_op2(UVaL_Reference<LegionWorkUnit> wu);

```

```

        // auxiliary member functions needed
        virtual void enable_functions
            (LegionInvocationStore *);
        virtual int invoke_method
            (UVaL_Reference<LegionWorkUnit> wu);
        virtual void accept_member_functions();
    };
#endif
%-----Simple.trans.c-----%
// The 'wrapper' class definition
#ifndef _C_Simple_trans_

#include <stdio.h>
#include <unistd.h>
#include "legion/Legion.h"
#include "Simple.trans.h"

// This is the wrapped object
static Legion_Simple *wrapper;

// This is the event handler for invoking a method
static int LegionMethodInvoke
    (UVaL_Reference<LegionEvent>);

// -----
// Generates a MethodDone event. Should be called
// right after the call to the wrapped object's
// member function.
void
Legion_Simple::
generate_MethodDoneEvent()
{
    UVaL_Reference<LegionEvent> methodDoneEvent;

    methodDoneEvent = new LegionEvent
        (LegionEvent_MethodDone, (void *) NULL);
    LegionEventManagerDefault.announce
        (methodDoneEvent, LegionEventAnnounceNow);
}

// -----
// Allocates the wrapped object and enables the
// corresponding functions in the invocation store.
Legion_Simple::
Legion_Simple()
{
    object = new Simple();
    wrapper = this;
    enable_functions(LegionInvocationStoreLL_Default);
}

// -----
// De-allocates the wrapped object
Legion_Simple::
~Legion_Simple()
{
    delete object;
    wrapper = NULL;
}

```

```

// -----
// wrapper member function for wrapped.op1()
void
Legion_Simple::
Legion_op1(UVaL_Reference<LegionWorkUnit> wu)
{
    // get the parameter from the work unit
    int parm1;
    UVaL_Reference<LegionBuffer> lb;
    lb = wu->get_parameter(1);
    lb->get_int(&parm1, 1);

    // Invoke the wrapped member function
    int result = object->op1(parm1);
    generate_MethodDoneEvent();

    // Return the result.
    UVaL_Reference<LegionBuffer> return_lb;
    return_lb = new LegionBuffer();
    return_lb->put_int(&result, 1);
    Legion_return(UVaL_METHOD_RETURN_VALUE,
        *(wu->get_continuation_list()), return_lb);
}

// -----
// wrapper member function for wrapped.op2()
void
Legion_Simple::
Legion_op2(UVaL_Reference<LegionWorkUnit> wu)
{
    int parm1, parm2;
    UVaL_Reference<LegionBuffer> lb;
    lb = wu->get_parameter(1);
    lb->get_int(&parm1, 1);
    lb = wu->get_parameter(2);
    lb->get_int(&parm2, 1);

    int result = object->op2(parm1, parm2);
    generate_MethodDoneEvent();

    // Return all results. In this case, there are
    // three of them (return value + 3 in/out
    // parameters).
    UVaL_Reference<LegionBuffer> return_lb;
    return_lb = new LegionBuffer();
    return_lb->put_int(&result, 1);
    Legion_return(UVaL_METHOD_RETURN_VALUE,
        *(wu->get_continuation_list()), return_lb);
    return_lb = new LegionBuffer();
    return_lb->put_int(&parm1, 1);
    Legion_return(1,
        *(wu->get_continuation_list()), return_lb);
    return_lb = new LegionBuffer();
    return_lb->put_int(&parm2, 1);
    Legion_return(2,
        *(wu->get_continuation_list()), return_lb);
}

```

```

// -----
// Invokes the appropriate method based on the
// function number in the supplied work unit.
int
Legion_Simple::
invoke_method(UVaL_Reference<LegionWorkUnit> wu)
{
    switch (wu->get_function_number()) {
        case SIMPLE_OP1_FUNCTION_NUMBER:
            Legion_op1(wu);
            break;
        case SIMPLE_OP2_FUNCTION_NUMBER:
            Legion_op2(wu);
            break;
        default:
            fprintf(stderr, "Legion_Simple::invoke_method()\n");
            fprintf(stderr, "This object does not export
            function number %d\n",
            wu->get_function_number());
            exit(0);
            break;
    }
}

// -----
// This is the server loop.
// EventMgr.serverLoop() continuously flushes events
// and then blocks waiting for events to become
// available.
void
Legion_Simple::
accept_member_functions()
{
    LegionEventManagerDefault.serverLoop();
}

// -----
// Enable the wrapped object's functions. The LIS
// must be explicitly told which functions to accept.
// There will eventually be some object mandatory
// functions in here too.
void
Legion_Simple::enable_functions(LegionInvocationStore
*LIS)
{
    // Enable the function numbers that I can handle...
    LegionInvocationStoreLL_Default->enable_function(
        SIMPLE_OP1_FUNCTION_NUMBER, DEFAULT_PRIORITY);
    LegionInvocationStoreLL_Default->enable_function(
        SIMPLE_OP2_FUNCTION_NUMBER, DEFAULT_PRIORITY);

    // Register my event handler...
    LegionEvent_MethodReady.addHandler
        (LegionMethodInvoke, 1.0);
}

```

```

// -----
// This is the event handler that get called on a
// MethodReady event. A MethodReady event is
// generated every time a ready invocation is
// inserted into the invocation store.
static int
LegionMethodInvoke(UVaL_Reference<LegionEvent> event)
{
    UVaL_Reference<LegionBuffer> lb;
    int parameter;

    if (LegionInvocationStoreLL_Default->any_ready()) {
        UVaL_Reference<LegionWorkUnit> wu;
        wu = LegionInvocationStoreLL_Default-
>next_matched();
        wrapper->invoke_method(wu);
    }
    return 0;
}

// -----
int
main (int argc, char **argv)
{
    Legion.init();
    wrapper = new Legion_Simple();
    Legion.AcceptMethods();
    wrapper->accept_member_functions();
}
#endif

```

6.3 ex1_Simple.c

```

%-----ex1_Simple.c-----%
#include <stdio.h>
#include "Simple.trans.h"
#include "legion/Legion.h"

UVaL_Reference<LegionParameter>
make_int_parameter(int parm_value, int parm_number)
{
    UVaL_Reference<LegionBuffer> lb;
    UVaL_Reference<LegionParameter> parm;

    lb = (LegionBuffer *) new LegionBuffer();
    lb->put_int(&parm_value, 1);
    parm = (LegionParameter *)
        new LegionParameter(parm_number, lb);
    return parm;
}

int
main(int argc, char **argv)
{
    // Variables for the 'user' code
    int a = 10, b = 15;
    int x, y, z;

```

```
// Initialize legion state
// All of the below to get a random instance number
int my_instance_number;
struct timeval tv;
gettimeofday(&tv, NULL);
srand(tv.tv_sec ^ tv.tv_usec);
my_instance_number =
    rand() ^ tv.tv_sec ^ tv.tv_usec;

// Initialize Legion Library
Legion.init();

// Manufacture my own LOID because I'm a command
// line object
Legion.SetMyLOID
(make_loid(UVaL_CLASS_ID_COMMANDLINE,
    my_instance_number));

// Tell my creator I'm ready to go
Legion.AcceptMethods();

// Create an empty program graph
LegionProgramGraph G(Legion.GetMyLOID());

// Create a couple of 'Simple' objects
UVaL_Reference<LegionLOID> A_name, B_name;
A_name = Legion.CreateObject
    (SIMPLE_OBJECT_CLASS_ID);
B_name = Legion.CreateObject
    (SIMPLE_OBJECT_CLASS_ID);

// Get handles for each object
LegionCoreHandle A_handle(A_name),
    B_handle(B_name);

// First call: x = A.op1(a);
// invoke()'s signature is
// invoke(function_num, num_parms, num_results);
UVaL_Reference<LegionInvocation> inv1;
inv1 = A_handle.invoke
    (SIMPLE_OP1_FUNCTION_NUMBER, 1, 1);
G.add_invocation(inv1);
UVaL_Reference<LegionParameter> parm1;
parm1 = make_int_parameter(a, 1);
G.add_constant_parameter(inv1, parm1, 1);

// Second call: y = B.op1(b);
UVaL_Reference<LegionInvocation> inv2;
inv2 = B_handle.invoke
    (SIMPLE_OP1_FUNCTION_NUMBER, 1, 1);
G.add_invocation(inv2);
UVaL_Reference<LegionParameter> parm2;
parm2 = make_int_parameter(b, 1);
G.add_constant_parameter(inv2, parm2, 1);
```

```
// Third call: z = A.op1(x, y);
// Both parameters are values yet to be computed,
// so they must be invocation parameters.
UVaL_Reference<LegionInvocation> inv3;
inv3 = A_handle.invoke
    (SIMPLE_OP2_FUNCTION_NUMBER, 2, 3);
G.add_invocation(inv3);
G.add_invocation_parameter
    (inv3, inv1, 1, UVaL_METHOD_RETURN_VALUE);
G.add_invocation_parameter
    (inv3, inv2, 2, UVaL_METHOD_RETURN_VALUE);

// We specifically ask for the in/out parameters.
// Don't get them otherwise.
G.add_result_dependency(inv3, 1);
G.add_result_dependency(inv3, 2);

// printf ("%d\n", z);
// We need 'z', so we must execute
G.execute();

// and wait for the return.
UVaL_Reference<LegionBuffer> lb;
lb = G.get_value (inv3, UVaL_METHOD_RETURN_VALUE);
lb->get_int(&z, 1);
printf ("z is%d\n", z);

// Since we asked for them, we can get the other
// values too.
G.release_all_values();
lb = G.get_value (inv3, 1);
lb->get_int(&x, 1);
printf ("x is %d\n", x);
lb = G.get_value (inv3, 2);
lb->get_int(&y, 1);
printf ("y is %d\n", y);
Legion.DestroyObject(A_name);
Legion.DestroyObject(B_name);
}
```

7.0 Interfaces

Below are selected parts of the interface from selected objects that make up the Legion Library implementation.

7.1 LegionProgramGraph

UVaL_Reference<LegionInvocation> add_invocation ()

Parameters: UVaL_Reference<LegionInvocation>

Add an invocation to the program graph. Returns the invocation if successful, NULL if not.

ParameterStatus add_constant_parameter()

Parameters: UVaL_Reference<LegionInvocation> target

UVaL_Reference<LegionParameter> parameter

int parameter_number

Adds the given parameter to the specified invocation as the 'parameter_number'th parameter. The parameter is a LegionParameter, which means that it contains an already computed value. The other possible is that it is an invocation parameter. This means that the parameter itself is a LegionInvocation, thus representing a computation that has yet to be performed.

ParameterStatus add_invocation_parameter()

Parameters: UVaL_Reference<LegionInvocation> source

UVaL_Reference<LegionInvocation> target

int source_parameter_number

int target_parameter_number

Adds the given source parameter as a parameter to the given target parameter. This call creates what is called a LegionContinuation, and adds the continuation to the continuation list for the source parameter. When an invocation request is eventually executed, the invokee must know where to send the results of the execution. Each LegionContinuation identifies a destination LOID to which a result should be sent.

void add_result_dependency()

Parameters: UVaL_Reference<LegionInvocation> source

int parameter_number

The current implementation always sends the return value of a method request back to the invoker. If the invoker wishes to receive other result values that e.g. might correspond to in/out parameters to the method invocation, then those values must be explicitly asked for using add_result_dependency. This call then creates a LegionContinuation corresponding to the requested parameter.

UVaL_Reference<LegionBuffer> get_value();

Parameters: UVaL_Reference<LegionInvocation> inv
 int parameter_number

Gets the specified return value for the given LegionInvocation out of the program graph. This is essentially a call through the program graph to the underlying message database. If the result is not available, get_value blocks until it is. A LegionBuffer is returned.

int execute();

Parameters: None

Takes the program graph rooted at the provided invocation and fire it off to be executed. Does not block.

7.2 LegionInvocationStore

int enable_function ();

Parameters: int func_num
 int priority

Enables the supplied function so that the LIS will accept method requests for it. Method requests for disabled functions are not accepted. High values for priority mean high priority.

int any_ready();

Parameters: None

Check to see if any method requests are ready.

int any_ready_for_func()

Parameters: int func_num;

Check to see if any method requests for the given function are ready.

UVaL_Reference<LegionWorkUnit> next_matched();

Parameters: None

Returns the next work unit. The priority scheme is obeyed.

UVaL_Reference<LegionWorkUnit> next_matched_for_func()

Parameters: int func_num;

Return the next work unit with the given function number.

int set_priority ();

Parameters: int func_num
 int priority

Set the priority for the given function number.

int insert();

Parameters: UVaL_Reference<LegionWorkUnit> new_work_unit

Insert the provided work unit into the invocation store. This work unit may be a method request or it may be a result from a previous method invocation.

UVaL_Reference<LegionWorkUnit> get_return_value();

Parameters: UVaL_Reference<LegionComputationTag> tag
int parameter_number

Returns the work unit with the given tag and parameter number. Typically, this function is called by a higher layer (e.g. LegionProgramGraph) which will unwrap the returned work unit to get the parameter inside.

int release_return_value();

Parameters: UVaL_Reference<LegionComputationTag> tag
int parameter_number

Deletes the work unit that matches the supplied tag/parameter_number pair.

int release_all_return_values();

Parameters: None
Deletes all return values from the invocation store.

7.3 LegionWorkUnit

int get_function_number();

Parameters: None
Returns the target function number for the work unit.

UVaL_Reference<LegionContinuationList> get_continuation_list();

Parameters: None
Return the continuation list for the results of this work unit.

UVaL_Reference<LegionBuffer> get_parameter();

Parameters: int parameter_number;
Returns the given parameter as a LegionBuffer. NULL if the parameter is not in the work unit.

7.4 LegionBuffer

7.4.1 Constructors

LegionBuffer()

Parameters: None
The default constructor is for the common case; it creates an empty buffer with a LegionStorageScat storage, and default implementations of the packer, compressor, and encryptor function sets.

LegionBuffer()

Parameters: UVaL_Reference<LegionStorage>
The simple constructor wraps a default buffer around a given storage object. It uses default implementations for the packer, encryptor, and compressor.

LegionBuffer()

Parameters: LegionMetaData metadata

The meta-data based constructor creates a new empty **LegionBuffer** with a LegionStorageScat default storage implementation. It instantiates packer, encryptor and compressors based on the meta-data.

LegionBuffer()

Parameters: UVaL_Reference<LegionStorage>

LegionMetaData metadata

The full featured constructor wraps a specified storage in a buffer appropriate for the given meta-data, selecting the right packer meta-data, compressor, and encryptor implementations.

7.4.2 Operations on the associated LegionStorage

A LegionStorage exports member functions to read and write untyped characters from and to a logical buffer. These operations are also exported by LegionBuffer, but the user should be warned that the “better” way to put data into a buffer is through the LegionPacker interface; read() and write() will not perform appropriate data format conversions, but put_int() and get_int() will.

size_t read()

Parameters: size_t num_bytes
void *data

Reads **num_bytes** bytes from the buffer starting at the current location of the buffer pointer. Copies the bytes into the space pointed to by data_dest. Returns the number of bytes actually read, and positions the buffer pointer immediately after the last byte that was read. Read() will not read past the end of the buffer.

size_t write()

Parameters: size_t num_bytes
void *data

Writes num_bytes bytes pointed to by data into the buffer starting at current position of the buffer pointer (overwriting existing data). Returns the number of bytes actually written, and positions the buffer pointer immediately after the last byte that was written. Writing past the end of the buffer causes it to expand.

size_t seek()

Parameters: seek_start whence
int bytes_away

Changes the position of the buffer pointer. The whence parameter can be BEGINNING (0), CURRENT (1), or END (2), and bytes_away tells how many bytes away from whence to set the pointer. Seeking past the end of the buffer causes it to

expand and to be filled with NULL bytes. Seeking to negative logical positions the buffer does not cause the buffer to expand; the buffer pointer is placed at logical position 0.

size_t size()

Parameters: None

Returns the current size of the buffer in bytes.

size_t current_byte()

Parameters: None

Returns the number of the byte to which the buffer pointer currently points. `current_byte()` returns 0 when the buffer pointer is at the beginning of the buffer, and `current_byte() == size()` when the buffer pointer is at the end of the buffer.

char *linearize()

Parameters: `int pack_metadata=0`

Returns a pointer to the beginning of the buffer's data. This pointer is guaranteed to point to data that is contiguous in memory. Depending on the implementation of the LegionStorage, this function may or may not need to return a pointer to a copy of the data.

void setMetaData()

Parameters: `LegionMetaData md`

Sets the metadata associated with the LegionStorage.

LegionMetaData getMetaData()

Parameters: None

Returns the meta data associated with the LegionStorage. This is useful for instantiating an appropriate LegionBuffer based on a given LegionStorage.

7.4.3 Operations on the associated LegionPacker

A LegionPacker exports operations for packing and unpacking the basic C++ data types into and out of a LegionBuffer in a particular data format. A LegionPacker exports `put_ZZZ()` and `get_ZZZ()` for all ZZZ in {char, short, ushort, int, long, ulong, float, double}.

size_t put_ZZZ()

Parameters: `ZZZ *source`

`int how_many`

Assumes that `source` points to an array of `how_many` instances of type ZZZ. Copies this data into the buffer after first performing the appropriate data conversion operation if necessary and appropriate for the type of LegionPacker that is instantiated.

size_t get_ZZZ()

Parameters: ZZZ *source
int how_many

Assumes that **source** points to enough space for an array of how_many instances of type ZZZ. Copies the next data from the LegionBuffer into this space after first performing the appropriate data conversion operation if necessary and appropriate for the type of LegionPacker that is instantiated.

7.4.4 Operations on the associated LegionEncryptor

Since no encryption algorithms have been implemented, the current encryption operations, `encrypt()` and `decode()`, are merely placeholders until the right set of encryption operations are defined.

7.4.5 Operations on the associated LegionCompressor

Since no compression algorithms have been implemented, the current compression operations, `compress()` and `decompress()`, are merely placeholders until the right set of compression operations are defined.

7.4.6 LegionPackable

int pack()

Parameters: int unpack()
LegionBuffer &lb
LegionBuffers are themselves packable.

7.4.7 Other functions

show()

Parameters: None
Prints the contents of the LegionBuffer to `stderr`. This is done however the associated LegionStorage sees fit.

7.4.8 Naming and binding: LegionLOID

LegionLOID is intended to be a base class for LOID's that enforce a particular structure on the fields of the LOID. An LOID contains four private data members, (1) an integer that holds the type of LOID, (2) an integer that indicates how many fields the LOID contains, (3) an array `field_size[]` of integers that holds the sizes in bytes of the LOID fields, and (4) an array `field_value[]` of pointers to the field data. Currently, the only derived class is called `LegionGeneralPurposeLOID`, which simply exposes the protected members to the public interface.

LegionLOID() (protected)

Parameters: int ltype

Sets the type to ltype, sets all other fields to zero.

LegionLOID() (protected)Parameters: int ltype,
short nfields

Sets the type to **ltype**. Sets num_fields to nfields. Allocates the field_size[] and field_value[] arrays. Sets all field_size[]'s to 0, sets all field_value[]'s to NULL.

LegionLOID() (protected)Parameters: int ltype,
short nfields,
short *fld_size

Sets the type to ltype. Sets num_fields to nfields. Allocates the field_size[] and field_value[] arrays. Sets all field_size[]'s to the values contained in the fld_size[] array. Allocates the field_value[] entries to the right size and zeros them out. This constructor assumes the fld_size array has at least nfields elements.

LegionLOID() (protected)Parameters: int ltype
short nfields
short *fld_size
char **fld_value

Sets the type to ltype. Sets num_fields to nfields. Allocates the field_size[] and field_value[] arrays. Sets all field_size[]'s to the values contained in the fld_size[] array. Allocates the field_value[] entries to the right size and copies the values from fld_value[] array into the field_value[] array. This constructor assumes that the fld_size[] and fld_value[] arrays have at least nfields elements, and that each fld_value[i] points to at least fld_size[i] bytes of space.

set_field_size() (protected)Parameters: short field_num
short fsize

Sets the appropriate field_size element to fsize and makes sure that the corresponding field_value element is at least **fsize** bytes. If it is not, it deletes the old field_value entry and allocates a new one. Only derived classes should be allowed to call this member.

set_type() (protected)

Parameters: int new_type

Sets the type entry to be new_type. Can only be called from within the code of derived classes.

LegionLOID()

Parameters: LegionBuffer &lb

LegionLOID()

Parameters: LegionLOID &otherLOID

These two constructors are public because neither allows the caller to violate the structure of any particular type of LOID – both just copy the LOID from the parameter, either a LegionBuffer or another LegionLOID.

LegionLOID()

Parameters: None

Not a very useful constructor, so it prints a warning and assigns all data members to zero. Useful constructors should at least say what the type is.

Accessors

Methods for getting and setting the type and all field values, by field number and field name, exist.

Overloaded operators

The ==, !=, and = operators are overloaded appropriately. An LOID is equal to another only if all fields are identical in size and value.

int is_empty()

Parameters: None

Returns 1 only if the LOID is of type UVaL_LegionLOID_type_EMPTY (zero).

int is_class()

Parameters: None

Returns 1 only if the LOID seems to refer to a class object, i.e. the instance number field is empty.

int same_class_as()

Parameters: UVaL_Reference<LegionLOID> other_loid

Returns 1 if the class_id field matches that of other_loid.

int pack()

LegionBuffer &lb

Type and num_fields are packed first, *in network order*. Next num_fields shorts are packed, *in network order*. Next, num_fields values are packed.

int unpack()

Parameters: LegionBuffer &lb

The type, num_fields, and field_size[]'s are unpacked into host order. The field_value[]'s are unpacked without switching the byte order.

int show()

Parameters: None

Prints the contents of the LOID to stderr for debugging purposes.

7.5 LegionMessage

LegionMessage()Parameters: UVaL_Reference<LegionLOID> src
UVaL_Reference<LegionLOID> dest

int fnum

int parms_to_expect

UVaL_Reference<LegionComputationTag> tag

UVaL_Reference<LegionParameterList> plist

UVaL_Reference<LegionContinuationList> lcontList

UVaL_Reference<LegionEnvironment> lenv

The LegionMessage constructor creates a LegionMessage from the constituent parts passed as parameters.

Accessor functions

LegionMessage exports public member functions to get and set all of its constituent parts.

Overloaded operators

The equality operators (== and !=) are overloaded. Two LegionMessages are deemed equal only if each of the constituent parts are equal, as determined by the equality operators of their respective classes.

show()

Parameters: None

Prints the contents of the LegionMessage to the stderr stream.

int pack()**int unpack()**

Parameters: LegionBuffer &lb

LegionMessage is packable.

7.5.1 LegionParameter

LegionParameter()

Parameters: int param_number

UVaL_Reference<LegionBuffer> lb

Constructs a new parameter whose value is assumed to be in lb, and whose number is set to param_number.

Other constructors

The default constructor creates a parameter with a negative parameter number and an empty LegionBuffer. A constructor that takes only a LegionBuffer as a parameter unpacks the contents of the LegionParameter from that buffer. The copy constructor is also overloaded.

Accessor functions

LegionParameter exports public member functions to get and set both the parameter number and the buffer containing the value of the parameter.

Overloaded operators

The == operator is overloaded to return 1 when the parameter numbers are the same, and 0 otherwise.

show()

Parameters: None

Prints the contents of the LegionParameter to the stderr stream.

int pack()**int unpack()**

Parameters: LegionBuffer &lb

LegionParameter is packable.

7.5.2 LegionParameterList

Constructors

The default constructor creates an empty parameter list, and a constructor that takes a LegionBuffer as an argument unpacks the contents of the LegionParameterList from that buffer.

Set operations

LegionParameterList is derived from templated class UVaL_PackableSet_LinkedList, and therefore exports the full interface of UVaL_PackableSet.

UVaL_Reference<LegionParameter> find()

Parameters: int parameter_number

Augments the UVaL_Set operations to allow parameters to be looked up by number. Returns a reference to the parameter, if found, or a null reference if not.

show()

Parameters: None

Prints the contents of the LegionParameterList to the stderr stream.

int pack()**int unpack()**

Parameters: LegionBuffer &lb
LegionParameterList is packable.

7.5.3 LegionComputationTag

LegionComputationTag simply maintains a glorified interface to a long integer. The Library also contains a class – LegionComputationTagGenerator – that creates random computation tags. See the source code or on-line documentation for a description of that class.

Constructors

The default constructor creates a LegionComputationTag with an uninitialized initial value. A constructor that takes a LegionBuffer as an argument unpacks the contents of the computation tag from that buffer.

Accessor functions

LegionComputationTag exports public member functions to allow the value of the tag to be set and retrieved as a long integer.

show()

Parameters: None

Prints the contents of the LegionComputationTag to the stderr stream.

int pack()**int unpack()**

Parameters: LegionBuffer &lb
LegionComputationTag is packable.

7.5.4 Other fields

A LegionMessage also contains a LegionContinuationList, and a LegionEnvironment. A LegionContinuationList is simply a UVal_PackableSet of LegionContinuations, and a LegionEnvironment is a UVal_PackableSet of LegionEnvironmentItems. Please refer to the on-line documentation and source code for the interface to these classes.

Legion Grid library

8.0 Grid library

The CLegionLib.h file contains the declarations for the C Library interface to the Legion system. This library is specifically designed to allow ease of use in many languages (including Fortran, C, C++). For additional functionality, the CError.h header file contains all necessary declarations to doing some small amount of error handling.

```

/* Change Log:          */
#ifndef CLEGION_LIB_H
#define CLEGION_LIB_H

#ifdef __cplusplus
extern "C" {
#endif

#include <stdio.h>

```

Note that anytime this header file talks about freeing memory, it is referring to using the C-language's `free()` function to return memory to the heap, NOT the C++ language's delete operators.

```

/* LOID Parameters */
#define LOID_PARAM      0
#define CONTEXT_PARAM  1

/* These defines give the
 * two valid values for
 * LegionIdType variables
 * parameters.
 */

typedef      int      LegionIdType;

/* This type is used in
 * function calls to indicate
 * whether or note the
 * parameter is a LOID. The
 * valid values are LOID_PARAM
 * and CONTEXT_PARAM.
 */

typedef      int      CLegionBinding;
typedef      int      CLegionHostReservation;
typedef      int      CLegionHostReservationRecord;
typedef      int      CLegionVaultReservation;
typedef      int      CLegionVaultReservationRecord;
typedef      int      CLegionReservation;
typedef      int      CLegionHostObjectStatus;
typedef      int      CLegionOPRAAddress;
typedef      int      CUVaL_InstanceRecord;
typedef      int      CInstancePlacementInfo;
typedef      int      CLegionCollectionData;

/* These typedefs are used to
 * represent all of the
 * necessary C++ classes that
 * would normally be used in
 * their place. In all cases
 * these are actually pointers
 * to LRefs to the actual
 * class type. This double
 * level of pointer
 * indirection was done to
 * allow for some small measure
 * of reference counting.
 */

```

The following section contains all of the valid Legion architectures.

```

#define LEGION_ARCH_linux          1
#define LEGION_ARCH_solaris        2
#define LEGION_ARCH_sun4           3
#define LEGION_ARCH_alpha_linux    4
#define LEGION_ARCH_alpha_DEC      5
#define LEGION_ARCH_sgi            6
#define LEGION_ARCH_rs6000         7
#define LEGION_ARCH_hp             8
#define LEGION_ARCH_t90            9
#define LEGION_ARCH_c90           10
#define LEGION_ARCH_t3e           11
#define LEGION_ARCH_x86_freebsd    12
#define LEGION_ARCH_winnt_x86     13

```

All programs that use the CLegionLib library contain some small amount of state which determines how the CLegionLib library interacts with certain Legion functions.

The following section gives the declarations for the functions that allow the user to interface with this state.

```

void SetTimeoutValue      (int Seconds); /* These functions set and clear */
void ClearTimeoutValue    ();           /* the default timeout value for */
                                   /* most Legion method invocations. */
void SetForceActivation   ();           /* In some small number of calls, */
void ClearForceActivation ();           /* a target Legion object may not */
                                   /* not yet be active. The force */
                                   /* activation flag indicates */
                                   /* whether the call should */
                                   /* activate the object or fail. */

```

All programs that use the CLegionLib library contain some small amount of state which determines how the CLegionLib library interacts with certain Legion functions. The following section gives the declarations for the functions that allow the user to interface with this state.

8.1 Legion Library interface

The following section lists the functions which interface with the C++ Legion Library.

int

Legion_init();

Initializes the legion library (for a Legion object started by a Legion host, not for a command line object). This function, or one of the other init functions must be called in all Legion programs before any other Legion calls are made. This does not preclude calls to the CLegionLib which do not directly interact with Legion.

Parameters: None

Return Values:

1 on Success
0 on Failure

User Responsibility: None

int

Legion_init_with_class(char *ClassID, int ClassIdLen);

Initializes the Legion library (for a Legion object started as an instance of a certain Legion class, but perhaps started on the user's command line). This function, or one of the other init functions must be called in all Legion programs before any other Legion calls are made. This does not preclude calls to the CLegionLib which do not directly interact with Legion.

Parameters:

char* ClassID = The representation of the class which instantiated the given object. These are not strings so much as byte arrays.

int ClassIdLen = Length of the "byte" array given for ClassID.

Return Values:

1 on Success
0 on Failure

User Responsibility: None

int

Legion_init_command_line_class();

Initializes the legion library (for a Legion object started on the command line as a main Legion program). For most client tools, this version of create will be the one used. This function, or one of the other init functions must be called in all Legion programs before any other Legion calls are made. This does not preclude

calls to the CLegionLib which do not directly interact with Legion.

Parameters: None

Return Values:
1 on Success
0 on Failure

User Responsibility: None

char*

Legion_GetLegionClassLOID();

Obtains the string representation of the LOID of the LegionClass object.

Parameters: None

Return Values:
LOID of the LegionClass object.

Error Return:
NULL

User Responsibility:
User is responsible for "free"ing memory allocated for the LOID.

CLegionBinding

Legion_GetLegionClassBinding();

Obtains the current binding for the LegionClass object.

Parameters: None

Return Values:
LegionBinding for LegionClass object

Error Return:
0

User Responsibility:
User is responsible for calling DestroyLegionBinding on the returned binding when done with it.

int

Legion_AcceptMethods();

Enables the receipt of messages in the Legion library. This function must be called prior to any other function which might make outcalls to other objects, or receive methods or return values from any other objects.

Parameters: None

Return Values:

1 on Success
0 on Failure

User Responsibility: None

int

Legion_DeleteSelf();

Indicates to the object's (program's) class that the program is finished and ready to go away. After this function is called, the user's code may run for some undetermined amount of time, but will eventually be killed if it doesn't exit.

Parameters: None

Return Values:

1 on Success
0 on Failure

User Responsibility: None

void

Legion_Sleep(int secs);

Causes the user's code to pause for approximately. the number of seconds requested, while still handling external Legion events (such as method receives, exception receives, etc.).

Parameters:

secs = Number of seconds to sleep for.

Return Values: None

User Responsibility: None

char*

ClassOf(LegionIdType IdType, char *ObjName);

Returns the LOID of the indicated object's class.

Parameters:

IdType = Either CONTEXT_PARAM or LOID_PARAM
 indicating what type of string name is given next.
ObjName = Either a LOID, or a context path to an object

Return Values:

LOID of the indicated object's class.

Error Return:

NULL

User Responsibility:

The user is responsible for "free"ing the LOID return value.

char*

GetMyLOID();

Returns the LOID of the running object(program).

Parameters: None

Return Values:

LOID of the current running program.

Error Return:

NULL

User Responsibility:

The user is responsible for "free"ing the LOID return value.

8.2 Exceptions

The following section describes functions which interface with the Legion Exception interface.

extern int CLEGION_CONTINUE_ON_EXCEPTION;	/* These variables correspond	*/
extern int CLEGION_EXIT_ON_EXCEPTION;	/* the valid values that can be	*/
extern int CLEGION_CANCEL_METHODS_ON_EXCEPTION;	/* used as flags when enabling	*/
extern int CLEGION_NO_MESSAGE_ON_EXCEPTION;	/* an exception catcher.	*/

void

Legion_ExceptionCatcherDefaultEnable(int Flag);

Enables a default Legion Exception Handler with user indicated properties.

Parameters:

Flag = A value from the valid flags listed above which indicates what the handler should do when a Legion exception is caught. Valid values include:

CLEGION_CONTINUE_ON_EXCEPTION
 CLEGION_EXIT_ON_EXCEPTION
 CLEGION_CANCEL_METHODS_ON_EXCEPTION
 CLEGION_NO_MESSAGE_ON_EXCEPTION
 (Don't do anything)

Return Values: None

User Responsibility: None

8.3 Legion context space

The following section provides a C interface to Legion context space.

int

ContextSpaceActive();

Determines whether or not a valid context space exists.

Parameters: None

Return Values:

0 if Context space is NOT active

1 if Context space is active

User Responsibility: None

ContextLookup(char *LookupPath);

Look up the LOID of a name in the current context (does not allow paths in the name).

Parameters:

Context name of an object to lookup in the current context.

Return Values:

LOID of the object indicated context space.

Error Return:

NULL

User Responsibility:

The user is responsible for "free"ing the LOID return value.

char*

ContextPathLookup(char *LookupPath);

Look up the LOID of the object named in the given "Unix-like" context path. This command will allow both absolute, and relative naming.

Parameters:

context path name of an object to lookup in context space.

Return Values:

LOID of the object indicated context space.

Error Return:

NULL

User Responsibility:

The user is responsible for "free"ing the LOID return value.

char**

ContextMultiLookup(char *LkupPath);

Look up the LOIDs of a number of objects named in the current context (not path specified, however). It's somewhat confusing as to why this command takes a string at all, but the string basically needs to contain "*" as its parameter.

Parameters:

Unclear, but give it ""

Return Values:

NULL terminated array of context space names for objects in the indicated context. In other words, each element of the array is a char*, which is either the context name of an object, or NULL if it is the end of the list.

Error Return:

NULL

User Responsibility:

The user is responsible for "free"ing each non-NULL entry in the array, as well as the array itself.

char**

ContextPathMultiLookup(char *LkupPath);

Look up the LOIDs of a number of objects named in a "Unix-like" context path (absolute or relative path names are perfectly valid here). It's somewhat confusing as to what the parameter to this command means. Basically, the thing to do here is to indicate the context path you want a listing for, and append "/" and "*" to it.

Parameters:

Unclear, but basically the path you want a listing of with "/" appended to it. I.e., if you want a listing of /home/mark, the parameter should be "/home/mark/ *".

(NOTE, in the above paragraph, there are spaces between / and * that should be removed. These are there to keep the C compiler from thinking we are commenting.)

Return Values:

NULL terminated array of context space names for objects in the indicated context. In other words, each element of the array is a char*, which is either the context name of an object, or NULL if it is the end of the list.

Error Return:

NULL

User Responsibility:

The user is responsible for “free”ing each non-NULL entry in the array, as well as the array itself.

int

ContextAdd(char *Name, char *Loid);

Add an entry to the current context. This new entry will have the indicated name, and will point to the object with the given LOID. This command assumes that the name is in the current context (not a full path).

Parameters:

Name = The name to add to the current context
Loid = The LOID of the object to link into context space.

Return Values:

1 on Success
0 on Failure

User Responsibility:

None

int

ContextRemove(char *Name);

Remove an entry from the current context. This entry must be in the current context. It cannot be a path name, but only a current context entry.

Parameters:

Name = The name to remove from context space.

Return Values:

1 on Success
0 on Failure

User Responsibility: None

int

ContextPathAdd(char *Name, char *Loid);

Add an entry to context space. This new entry will have the indicated name, and will point to the object with the given LOID. This command allows relative and absolute paths in context space to be added.

Parameters:

Name = The name to path to add to context space
LOID = The LOID of the object to link into context space.

Return Values:

1 on Success
0 on Failure

User Responsibility: None

int

ContextPathRemove(*char *Name*);

Remove a path from context space.

Parameters:

 Name = The path to remove from context space.

Return Values:

 1 on Success

 0 on Failure

User Responsibility: None

char*

ContextSingleReverseLookup(

*LegionIdType IdType, char *ObjName*);

Try and obtain a context path name for a given object. This command uses the object's attributes to determine this name and only returns the first name found. Note that the attribute database may or may not actually reflect the object's actual context space names

Parameters:

 IdType = Indicates whether the ObjName gives a context path name or a LOID.

 ObjName = Either the context name, or the LOID of the object to reverse lookup.

Return Values:

 First context path name of the object looked up.

Error Return:

 NULL

User Responsibility:

 The user is responsible for "free"ing the name returned.

char**

ContextMultiReverseLookup(

*LegionIdType, char *ObjName, int MaxNum*);

Try and obtain a list of context path names for a given object. This command uses the object's attributes to determine these names and only returns the first "MaxNum" entries found. Note that the attribute database may or may not actually reflect the object's actual context space names.

Parameters:

IdType = Indicates whether the ObjName gives a context path name or a LOID.
ObjName = Either the context name, or the LOID of the object to reverse lookup.
MaxNum = Maximum number of entries to return.

Return Values:

First "MaxNum" context path names of the object. These paths are stored in a NULL terminated array of strings.

Error Return:

NULL

User Responsibility:

The user is responsible for "free"ing each name returned in the list well as the list itself.

char*

ContextPWD();

Return the context path name of the current context.

Parameters: None

Return Values:

context path name of the current context.

Error Return:

NULL

User Responsibility:

The user is responsible for "free"ing the context path name returned.

char*

legion_mkdir(char *PathName);

Creates a new context named by the given path.

Parameters:

PathName = The new context's path name.

Return Values:

LOID of the new context that was created.

Error Return:

NULL

User Responsibility:

The user is responsible for "free"ing the context LOID.

int

legion_cd(*char *PathName*);

Change the current context to the one named in the path. NOTE that this only changes the current context for the running program, not for the shell or program that executed the program.

Parameters:

Pathname of the context to change to.

Return Values:

1 on Success

0 on Failure

User Responsibility: None

8.4 Legion object management

The following section contains functions which are used to manage actual Legion Objects (create, activate, destroy, etc.).

int

ObjectActive(*LegionIdType, char *Name*);

Indicates whether the given object is currently active or inert.

Parameters:

IdType = Flag indicating whether parameter is a
 LOID or a context path name.

Name = Either a context path name or a LOID.

Return Values:

1 if the object is active

0 if the object is inert

Error Return:

-1

User Responsibility: None

char*

Ping(*LegionIdType, char *Name*);

Ping a given object to see if it is responsive.

Parameters:

IdType = Flag indicating whether parameter is a
 LOID or a context path name.

Name = Either a context path name or a LOID.

Return Values:

LOID of the object pinged.

Error Return:
NULL

User Responsibility:
User is responsible for "free"ing the LOID returned.

char**

Create(*int NumObjects, LegionIdType, char* ClassName*);

Create a vector of new Legion objects by vector instantiating the given class.

Parameters:

NumObjects = Number of object to vector instantiate.
IdType = Flag indicating whether the ClassName parameter is a context path name or a LOID.
ClassName = LOID or context path name of the class to instantiate.

Return Values:

NULL terminated array of LOIDs indicating the objects that were instantiated. Each element of the array is a char* which is the LOID of one object created.

Error Return:
NULL

User Responsibility:
User is responsible for "free"ing each LOID in the array as well as the array itself.

char**

ScheduledCreate(*int NumObjects, LegionIdType CIDType, char* ClassName, LegionIdType SIDType, char *SchedName*);

Create a vector of new Legion objects by vector instantiating the given class. Used the given scheduler to schedule this vector instantiation.

Parameters:

NumObjects = Number of object to vector instantiate.
CIDType = Flag indicating whether the ClassName parameter is a context path name or a LOID.
ClassName = LOID or context path name of the class to instantiate.
SIDType = Flag indicating whether the SchedName parameter is a context path name or a LOID.
ClassName = LOID or context path name of the Scheduler to use when instantiating the class.

Return Values:

NULL terminated array of LOIDs indicating the objects that were instantiated. Each element of the array is a char* which is the LOID of one object created.

Error Return:

NULL

User Responsibility:

User is responsible for “free”ing each LOID in the array as well as the array itself.

char**

CreateRestricted(*int NumObjects,*
*LegionIdType CIDType, char *ClassName,*
*LegionIdType HIDType, char *HostName,*
*LegionIdType VIDType, char *VaultName,*
*LegionIdType IIDType, char *ImplName);*

Create a vector of new Legion objects by vector instantiating the given class. The user also indicates any combination of restrictions which may include restrictions to hosts, vaults or implementations.

Parameters:

NumObjects = Number of object to vector instantiate.

CIDType = Flag indicating whether the ClassName parameter is a context path name or a LOID.

ClassName = LOID or context path name of the class to instantiate.

HIDType = Flag indicating whether the HostName parameter is a context path name or a LOID.

HostName = LOID or context path name of the Host to which the instantiation is restricted.

VIDType = Flag indicating whether the HostName parameter is a context path name or a LOID.

VaultName = LOID or context path name of the Vault to which the instantiation is restricted.

IIDType = Flag indicating whether the HostName parameter is a context path name or a LOID.

ImplName = LOID or context path name of the implementation to which the instantiation is restricted.

Return Values:

NULL terminated array of LOIDs indicating the objects that were instantiated. Each element of the array is a char* which is the LOID of one object created.

Error Return:

NULL

User Responsibility:

User is responsible for “free”ing each LOID in the array as well as the array itself.

char**

ScheduledCreateRestricted(*int NumObjects,*
*LegionIdType CIDType, char *ClassName,*
*LegionIdType HIDType, char *HostName,*
*LegionIdType VIDType, char *VaultName,*
*LegionIdType IIDType, char *ImplName,*
*LegionIdType SIDType, char *SchedName*);

Create a vector of new Legion objects by vector instantiating the given class. The user also indicates any combination of restrictions which may include restrictions to hosts, vaults or implementations. This command also uses the given scheduler to schedule the instantiation.

Parameters:

NumObjects = Number of object to vector instantiate.

CIDType = Flag indicating whether the ClassName parameter is a context path name or a LOID.

ClassName = LOID or context path name of the class to instantiate.

HIDType = Flag indicating whether the HostName parameter is a context path name or a LOID.

HostName = LOID or context path name of the Host to restrict the instantiation to.

VIDType = Flag indicating whether the HostName parameter is a context path name or a LOID.

VaultName = LOID or context path name of the Vault to which the instantiation is restricted.

IIDType = Flag indicating whether the HostName parameter is a context path name or a LOID.

ImplName = LOID or context path name of the Implementation which the instantiation is restricted.

SIDType = Flag indicating whether the SchedName parameter is a context path name or a LOID.

ImplName = LOID or context path name of the Scheduler to use when scheduling the instantiation.

Return Values:

NULL terminated array of LOIDs indicating the objects that were instantiated. Each element of the array is a char* which is the LOID of one object created.

Error Return:

NULL

User Responsibility:

User is responsible for “free”ing each LOID in the array as well as the array itself.

char**

```
ComplxCreateRestricted(int NumObjects,  
    LegionIdType CIDType, char *ClassName,  
    LegionIdType HIDType, char **HostName,  
    LegionIdType VIDType, char **VaultName,  
    LegionIdType IIDType, char **ImplName);
```

Create a vector of new Legion objects by vector instantiating the given class. The user also indicates any combination of restrictions which may include restrictions to hosts, vaults or implementations.

Parameters:

NumObjects = Number of object to vector instantiate.

CIDType = Flag indicating whether the ClassName parameter is a context path name or a LOID.

ClassName = LOID or context path name of the class to instantiate.

HIDType = Flag indicating whether the HostName parameter is a NULL terminated list of context path names or a NULL terminated list of LOIDs.

HostName = NULL terminated array of LOIDs or context path names of the Hosts to which the instantiation is limited.

VIDType = Flag indicating whether the VaultName parameter is a NULL terminated list of context path names or a NULL terminated list of LOIDs.

VaultName = NULL terminated array of LOIDs or context path names of the Vaults to which the instantiation is restricted.

IIDType = Flag indicating whether the ImplName parameter is a NULL terminated list of context path names or a NULL terminated list of LOIDs.

ImplName = NULL terminated array of LOIDs or context path names of the Impls to which the instantiation is restricted.

Return Values:

NULL terminated array of LOIDs indicating the objects that were instantiated. Each element of the array is a char* which is the LOID of one object created.

Error Return:

NULL

User Responsibility:

User is responsible for “free”ing each LOID in the array as well as the array itself.

char**

ComplxScheduledCreateRestricted(*int NumObjects,*
*LegionIdType CIDType, char *ClassName,*
*LegionIdType HIDType, char **HostName,*
*LegionIdType VIDType, char **VaultName,*
*LegionIdType IIDType, char **ImplName,*
*LegionIdType SIDType, char *SchedName*);

Create a vector of new Legion objects by vector instantiating the given class. The user also indicates any combination of restrictions which may include restrictions to hosts, vaults or implementations. In addition, the user may specify a scheduler to use for the instantiation

Parameters:

NumObjects = Number of object to vector instantiate.

CIDType = Flag indicating whether the ClassName parameter is a context path name or a LOID.

ClassName = LOID or context path name of the class to instantiate.

HIDType = Flag indicating whether the HostName parameter is a NULL terminated list of context path names, or a NULL terminated list of LOIDs.

HostName = NULL terminated array of LOIDs or context path names of the Hosts to which the instantiation is restricted.

VIDType = Flag indicating whether the VaultName parameter is a NULL terminated list of context path names, or a NULL terminated list of LOIDs.

VaultName = NULL terminated array of LOIDs or context path names of the Vaults to which the instantiation is restricted.

IIDType = Flag indicating whether the ImplName parameter is a NULL terminated list of context path names, or a NULL terminated list of LOIDs.

ImplName = NULL terminated array of LOIDs or context path names of the Impls to which the instantiation is restricted.

SIDType = Flag indicating whether the SchedName parameter is a context path name or a LOID.

SchedName = Context path name or LOID of the scheduler to use when instantiating the class.

Return Values:

NULL terminated array of LOIDs indicating the objects that were instantiated. Each element of the array is a char* which is the LOID of one object created.

Error Return:

NULL

User Responsibility:

User is responsible for “free”ing each LOID in the array as well as the array itself.

int

Activate(*LegionIdType IType, char *ObjectName*);

Activate an inert Legion object

Parameters:

IdType = Flag indicating whether the ObjectName
ObjectName = LOID or context path name of the object to
 activate.

Return Values:

1 on Success
0 on failure

User Responsibility: None

int

ScheduledActivate(*LegionIdType OIDType, char *ObjectName,
 LegionIdType SIDType, char *SchedName*);

Activate an inert Legion object (using the indicated scheduler to schedule the activation)

Parameters:

OIDType = Flag indicating whether the ObjectName
 parameter is a context path name or a LOID.
ObjectName = LOID or context path name of the object to
 activate.
SIDType = Flag indicating whether the SchedName
 parameter is a context path name or a LOID.
SchedName = LOID or context path name of the scheduler
 to use when doing the activation.

Return Values:

1 on Success
0 on failure

User Responsibility: None

int

ActivateRestricted(*LegionIdType* *OIDType*, *char* **ObjectName*,
LegionIdType *HIDType*, *char* **HostName*,
LegionIdType *VIDType*, *char* **VaultName*,
LegionIdType *IIDType*, *char* **ImplName*);

Activate an inert Legion object, but subject to the restrictions indicated by the user.

Parameters:

OIDType = Flag indicating whether the *ObjectName* parameter is a context path name or a LOID.

ObjectName = LOID or context path name of the object to activate.

HIDType = Flag indicating whether the *HostName* parameter is a context path name or a LOID.

HostName = LOID or context path name of the Host to which the activation is restricted.

VIDType = Flag indicating whether the *VaultName* parameter is a context path name or a LOID.

VaultName = LOID or context path name of the Vault to which the activation is restricted.

IIDType = Flag indicating whether the *ImplName* parameter is a context path name or a LOID.

ImplName = LOID or context path name of the Impl to which the activation is restricted.

Return Values:

1 on Success

0 on Failure

User Responsibility: None

int

ScheduledActivateRestricted(
LegionIdType *OIDType*, *char* **ObjectName*,
LegionIdType *HIDType*, *char* **HostName*,
LegionIdType *VIDType*, *char* **VaultName*,
LegionIdType *IIDType*, *char* **ImplName*,
LegionIdType *SIDType*, *char* **SchedName*);

Activate an inert Legion object, but subject to the restrictions indicated by the user. Also, use the indicated scheduler to schedule the activation.

Parameters:

OIDType = Flag indicating whether the *ObjectName* parameter is a context path name or a LOID.

ObjectName = LOID or context path name of the object to activate.

HldType = Flag indicating whether the HostName parameter is a context path name or a LOID.
 HostName = LOID or context path name of the Host to which the activation is restricted.
 VIDType = Flag indicating whether the VaultName parameter is a context path name or a LOID.
 VaultName = LOID or context path name of the Vault to which the activation is restricted.
 IIDType = Flag indicating whether the ImplName parameter is a context path name or a LOID.
 ImplName = LOID or context path name of the Impl to which the activation is restricted.
 SIDType = Flag indicating whether the SchedName parameter is a context path name or a LOID.
 SchedName = LOID or context path name of the Scheduler to use when doing the activation.

Return Values:

1 on Success
 0 on failure

User Responsibility: None

int

ComplxActivateRestricted(

*LegionIdType OIDType, char *ObjectName,*
*LegionIdType HIDType, char **HostName,*
*LegionIdType VIDType, char **VaultName,*
*LegionIdType IIDType, char **ImplName);*

Activate an inert Legion object, but subject to the restrictions indicated by the user.

Parameters:

OIDType = Flag indicating whether the ObjectName parameter is a context path name or a LOID.
 ObjectName = LOID or context path name of the object to activate.
 HldType = Flag indicating whether the HostName parameter is a list of context path names or LOIDs.
 HostName = A NULL terminated array of LOIDs or context path names of Hosts to which the activation is restricted.
 VldType = Flag indicating whether the VaultName parameter is a list of context path names or LOIDs.
 VaultName = A NULL terminated array of LOIDs or context path names of Vaults to which the activation is restricted.

IldType = Flag indicating whether the **ImplName** parameter is a list of context path names or LOIDs.

ImplName = A NULL terminated array of LOIDs or context path names of Implementations to which the activation is restricted.

Return Values:

1 on Success
0 on failure

User Responsibility: None

int

ComplxScheduledActivateRestricted(

LegionIdType *OIDType*, *char *ObjectName*,
LegionIdType *HIDType*, *char **HostName*,
LegionIdType *VIDType*, *char **VaultName*,
LegionIdType *IIDType*, *char **ImplName*,
LegionIdType *SIDType*, *char *SchedName*);

Activate an inert Legion object, but subject to the restrictions indicated by the user and using the given scheduler.

Parameters:

OIDType = Flag indicating whether the **ObjectName** parameter is a context path name or a LOID.

ObjectName = LOID or context path name of the object to activate.

HIdType = Flag indicating whether the **HostName** parameter is a list of context path names or LOIDs.

HostName = A NULL terminated array of LOIDs or context path names of Hosts to which the activation is restricted.

VIdType = Flag indicating whether the **VaultName** parameter is a list of context path names or LOIDs.

VaultName = A NULL terminated array of LOIDs or context path names of Vaults to which the activation is restricted.

IldType = Flag indicating whether the **ImplName** parameter is a list of context path names or LOIDs.

ImplName = A NULL terminated array of LOIDs or context path names of Implementations to which the activation is restricted.

SIDType = A flag indicating whether the **SchedName** parameter is a context path name or a LOID.

SchedName = A context path name or LOID indicating the scheduler to use when doing the activation.

Return Values:

1 on Success
0 on failure

User Responsibility: None

int

DeleteObject(*LegionIdType IdType, char *ObjName*);

Deletes or Destroys a given object by asking its class to get destroy it. This function is identical to DestroyObject

Parameters:

IdType = Flag indicating whether parameter is a LOID
 or a context path name.
Name = Either the context path name or the LOID of
 the object to delete.

Return Values:

1 on Success
0 on Failure

User Responsibility: None

int

DestroyObject(*LegionIdType IdType, char *ObjName*);

Deletes or Destroys a given object by asking its class to get destroy it. This function is identical to DeleteObject

Parameters:

IdType = Flag indicating whether parameter is a
 LOID or a context path name.
Name = Either the context path name or the LOID of
 the object to destroy.

Return Values:

1 on Success
0 on Failure

User Responsibility: None

int

DeactivateObject(*LegionIdType IdType, char *ObjName*);

Deactivates a given object.

Parameters:

IdType = Flag indicating whether parameter is a
 LOID or a context path name.
Name = Either the context path name or the LOID of
 the object to deactivate.

Return Values:

1 on Success
0 on Failure

User Responsibility: None

int

ForcedDeactivateObject(*LegionIdType IdType, char *ObjName*);
Deactivates a given object. The forced in this call means that if the object is going up or going down, then an error is returned.

Parameters:

IdType = Flag indicating whether parameter is a
 LOID or a context path name.
Name = Either the context path name or the LOID of
 the object to deactivate.

Return Values:

1 on Success
0 on Failure

User Responsibility: None

int

GetObjectType(*LegionIdType IdType, char *ObjName*);
Returns an integer which indicates as specifically as possible what type of object the user is talking about.

Parameters:

IdType = Flag indicating whether parameter is a
 LOID or a context path.
ObjName = Either the context path name or the LOID of
 the object to be described.

Return Values:

Returns an integer indicating the type of the object (from the list above).

User Responsibility: None

```
#define  LEGION_OBJ_CONTEXT      1      /* These constants are used */
#define  LEGION_OBJ_FILE        2      /* to indicate what type of */
#define  LEGION_OBJ_RUNNABLE    3      /* object one is talking    */
#define  LEGION_OBJ_CLASS       4      /* about. These are the     */
#define  LEGION_OBJ_HOST        5      /* values that can be       */
#define  LEGION_OBJ_VAULT       6      /* by GetObjectType().      */
#define  LEGION_OBJ_TTY         7
#define  LEGION_OBJ_IMPLEMENTATION 8
```

```

#define  LEGION_OBJ_IMPLEMENTATION_CACHE    9
#define  LEGION_OBJ_LEGION_CLASS            10
#define  LEGION_OBJ_BINDING_AGENT           11
#define  LEGION_OBJ_MPI                     12
#define  LEGION_OBJ_PVM3                    13
#define  LEGION_OBJ_UNKNOWN                  14

```

int

isContext(*LegionIdType IdType, char *ObjName*);

Determines if the indicated object is a context object or not.

Parameters:

IdType = Flag indicating whether parameter is a
 LOID or a context path name.

ObjName = The context path name or the LOID of the
 object to test.

Return Values:

1 if the object is a context
0 if the object is not a context

User Responsibility: None

int

isFile(*LegionIdType IdType, char *ObjName*);

Determines if the indicated object is a File object or not.

Parameters:

IdType = Flag indicating whether parameter is a
 LOID or a context path name.

ObjName = The context path name or the LOID of the
 object to test.

Return Values:

1 if the object is a File
0 if the object is not a File

User Responsibility: None

int

isRunnable(*LegionIdType IdType, char *ObjName*);

Determines if the indicated object is a Runnable object or not.

Parameters:

IdType = Flag indicating whether parameter is a
 LOID or a context path name.

ObjName = Either the context path name or the LOID of
 the object to test.

Return Values:

- 1 if the object is a Runnable
- 0 if the object is not a Runnable

User Responsibility: None

int

isClass(*LegionIdType IdType, char *ObjName*);

Determines if the indicated object is a Class object or not.

Parameters:

- IdType = Flag indicating whether parameter is a
 LOID or a context path name.
- ObjName = The context path name or the LOID of the
 object to test.

Return Values:

- 1 if the object is a Class
- 0 if the object is not a Class

User Responsibility: None

int

isHost(*LegionIdType IdType, char *ObjName*);

Determines if the indicated object is a Host object or not.

Parameters:

- IdType = Flag indicating whether parameter is a
 LOID or a context path name.
- ObjName = The context path name or the LOID of the
 object to test.

Return Values:

- 1 if the object is a Host
- 0 if the object is not a Host

User Responsibility: None

int

isVault(*LegionIdType IdType, char *ObjName*);

Determines if the indicated object is a Vault object or not.

Parameters:

- IdType = Flag indicating whether parameter is a
 LOID or a context path name.
- ObjName = The context path name or the LOID of the
 object to test.

Return Values:

- 1 if the object is a Vault
- 0 if the object is not a Vault

User Responsibility: None

int

isTTY(*LegionIdType IdType, char *ObjName*);

Determines if the indicated object is a TTY object or not.

Parameters:

IdType =	Flag indicating whether parameter is a LOID or a context path name.
ObjName =	The context path name or the LOID of the object to test.

Return Values:

1 if the object is a TTY
0 if the object is not a TTY

User Responsibility: None

int

isImplementation(*LegionIdType IdType, char *ObjName*);

Determines if the indicated object is a Implementation object or not.

Parameters:

IdType =	Flag indicating whether parameter is a LOID or a context path name.
ObjName =	The context path name or the LOID of the object to test.

Return Values:

1 if the object is a Implementation
0 if the object is not a Implementation

User Responsibility: None

int

isImplementationCache(*LegionIdType IdType, char *ObjName*);

Determines if the indicated object is a ImplementationCache object or not.

Parameters:

IdType =	Flag indicating whether parameter is a LOID or a context path name.
ObjName =	The context path name or the LOID of the object to test.

Return Values:

1 if the object is a ImplementationCache
0 if the object is not a ImplementationCache

User Responsibility: None

int

isLegionClass(*LegionIdType IdType, char *ObjName*);

Determines if the indicated object is a LegionClass object or not.

Parameters:

IdType =	Flag indicating whether parameter is a LOID or a context path name.
ObjName =	The context path name or the LOID of the object to test.

Return Values:

1 if the object is a LegionClass
0 if the object is not a LegionClass

User Responsibility: None

int

isBindingAgent(*LegionIdType IdType, char *ObjName*);

Determines if the indicated object is a BindingAgent object or not.

Parameters:

IdType =	Flag indicating whether parameter is a LOID or a context path name.
ObjName =	The context path name or the LOID of the object to test.

Return Values:

1 if the object is a BindingAgent
0 if the object is not a BindingAgent

User Responsibility: None

int

isMPI(*LegionIdType IdType, char *ObjName*);

Determines if the indicated object is a MPI object or not.

Parameters:

IdType =	Flag indicating whether parameter is a LOID or a context path name.
ObjName =	The context path name or the LOID of the object to test.

Return Values:

1 if the object is a MPI
0 if the object is not a MPI

User Responsibility: None

int

isPVM3(*LegionIdType IdType, char *ObjName*);

Determines if the indicated object is a PVM3 object or not.

Parameters:

IdType = Flag indicating whether parameter is a
 LOID or a context path name.
ObjName = The context path name or the LOID of the
 object to test.

Return Values:

1 if the object is a PVM3
0 if the object is not a PVM3

User Responsibility: None

8.5 Legion attributes

The following functions deal with the attribute database that all Legion objects support.

int

AddAttribute(

*LegionIdType IdType, char *ObjName, char *Attribute*);

Add an attribute to an object's attribute database.

Parameters:

IdType = Flag indicating whether parameter is a LOID
 or a context path name.
ObjName = The context path name or the LOID of the
 object to which the attribute is to be added.
Attribute = The string representation of the attribute to
 add to the database. This is of the form:
 attribute_signature(attribute_value).

Return Values:

1 on Success
0 on Failure

User Responsibility: None

int

AddAttributes(

*LegionIdType IdType, char *ObjName, char **Attribute*);

Add a list of attributes to an object's attribute database.

Parameters:

IdType = Flag indicating whether parameter is a LOID or a context path name.

ObjName = The context path name or the LOID of the object to which the attribute will be added.

Attribute = A NULL terminated list of the string representations of the attributes to add to the database. These are of the form:
attribute_signature(attribute_value).

Return Values:

1 on Success
0 on Failure

User Responsibility: None

int

ReplaceAttribute(*LegionIdType IdType,*
*char *ObjName, char *OldAttribute, char *NewAttribute*);
Replace an attribute in an object's attribute database.

Parameters:

IdType = Flag indicating whether parameter is a LOID or a context path name.

ObjName = The context path name or the LOID of the object to which the attribute will be added.

OldAttribute = The string representation of the old attribute to replace in the database. This is of the form:
attribute_signature(attribute_value).

NewAttribute = The string representation of the new attribute to put into the database. This is of the form:
attribute_signature(attribute_value).

Return Values:

1 on Success
0 on Failure

User Responsibility: None

int

ReplaceAttributeSig(*LegionIdType IdType,*
*char *ObjName, char *OldAttributeSig, char *NewAttribute*);
Replace an attribute in an object's attribute database.

Parameters:

IdType = Flag indicating whether parameter is a LOID or a context path name.

ObjName = The context path name or the LOID of the object to which the attribute will be added.

OldAttributeSig = The string representation of the old attribute's attribute signature.

NewAttribute = The string representation of the new attribute to put into the database. This is of the form:

attribute_signature(attribute_value).

Return Values:

1 on Success

0 on Failure

User Responsibility: None

int

ReplaceAttributes(*LegionIdType IdType,*
*char *ObjName, char **OldAttributes, char **NewAttributes*);

Replace a list of attributes in an object's attribute database.

Parameters:

IdType = Flag indicating whether parameter is a LOID or a context path name.

ObjName = The context path name or the LOID of the object to which the attribute will be added.

OldAttributes = A NULL terminated list of the string representations of the old attributes to replace in the database. These are of the form:

attribute_signature(attribute_value).

NewAttributes = A NULL terminated list of the string representations of the new attributes to put into the database. These are of the form:

attribute_signature(attribute_value).

Return Values:

1 on Success

0 on Failure

User Responsibility: None

int

ReplaceAttributeSigs(*LegionIdType IdType, char *ObjName,*
*char **OldAttributeSigs, char **NewAttributes*);

Replace a list of attributes in an object's attribute database.

Parameters:

IdType = Flag indicating whether parameter is a LOID or a context path name.

ObjName = The context path name or the LOID of the object to which the attribute will be added.

OldAttributeSigs = A NULL terminated list of the string representations of the old attribute signatures to replace in the database.

NewAttributes = A NULL terminated list of the string representations of the new attributes to put into the database. These are of the form:

attribute_signature(attribute_value).

Return Values:

1 on Success

0 on Failure

User Responsibility: None

int

RemoveAttribute(

*LegionIdType IdType, char *ObjName, char *Attribute);*

Remove an attribute in an object's attribute database.

Parameters:

IdType = Flag indicating whether parameter is a LOID or a context path name.

ObjName = The context path name or the LOID of the object to which the attribute will be added.

Attribute = The string representation of the attribute to remove from the object's attribute database.

Return Values:

1 on Success

0 on Failure

User Responsibility: None

int

RemoveAttributeSig(

*LegionIdType IdType, char *ObjName, char *AttributeSig);*

Remove an attribute in an object's attribute database.

Parameters:

IdType = Flag indicating whether parameter is a LOID or a context path name.

ObjName = The context path name or the LOID of the object to which the attribute will be added.

Attribute = The string representation of the signature of the attribute to remove from the object's attribute database.

Return Values:

1 on Success
0 on Failure

User Responsibility: None

int

RemoveAttributes(

*LegionIdType IdType, char *ObjName, char **Attributes);*

Remove a set of attributes from an object's attribute database.

Parameters:

IdType =	Flag indicating whether parameter is a LOID or a context path name.
ObjName =	The context path name or the LOID of the object to which the attribute will be added
Attributes =	A NULL terminated list of the string representations of the attributes to remove from the object's attribute database.

Return Values:

1 on Success
0 on Failure

User Responsibility: None

int

RemoveAttributeSigs(

*LegionIdType IdType, char *ObjName, char **AttributeSigs);*

Remove a set of attributes from an object's attribute database.

Parameters:

IdType =	Flag indicating whether parameter is a LOID or a context path name.
ObjName =	The context path name or the LOID of the object to which the attribute will be added.
AttributeSigs =	A NULL terminated list of the string representations of the signatures of attributes to remove from the object's attribute database.

Return Values:

1 on Success
0 on Failure

User Responsibility: None

char*

RetrieveAttribute(*LegionIdType IdType*,
 *char *ObjName*,
 *char *Attribute*);

Retrieve an attribute from an object's attribute database.

Parameters:

IdType =	Flag indicating whether parameter is a LOID or a context path name.
ObjName =	The context path name or the LOID of the object to which the attribute will be added.
Attribute =	The string representation of the attribute to retrieve from the object's attribute database.

Return Values:

The string representation of the attribute retrieved.

Error Return:

NULL

User Responsibility:

The user is responsible for "free"ing the returned attribute.

char**

RetrieveAttributes(*LegionIdType IdType*,
 *char *ObjName*,
 *char **Attributes*);

Retrieve a set of attributes from an object's attribute database.

Parameters:

IdType =	Flag indicating whether parameter is a LOID or a context path name.
ObjName =	The context path name or the LOID of the object to which the attribute will be added.
Attributes =	A NULL terminated list of the string representations of the attributes to retrieve from the object's attribute database.

Return Values:

A NULL terminated list of the string representations of the attributes retrieved.

Error Return:

NULL

User Responsibility:

The user is responsible for "free"ing each returned attribute in the list as well as for "free"ing the list itself.

char*

RetrieveAttributeSig(

*LegionIdType IdType, char *ObjName, char *AttributeSig);*

Retrieve an attribute from an object's attribute database.

Parameters:

IdType = Flag indicating whether parameter is a LOID or a context path name.
ObjName = Either the context path name or the LOID of the object to which the attribute will be added.
AttributeSig = The string representation of the signature of the attribute to retrieve from the object's attribute database.

Return Values:

The string representation of the attribute retrieved.

Error Return:

NULL

User Responsibility:

The user is responsible for "free"ing the returned attribute.

char**

RetrieveAttributeSigs(

*LegionIdType IdType, char *ObjName, char **AttributeSigs);*

Retrieve a list of attributes from an object's attribute database.

Parameters:

IdType = Flag indicating whether parameter is a LOID or a context path name.
ObjName = The context path name or the LOID of the object to which the attribute will be added.
AttributeSigs = A NULL terminated list of the string representations of the signatures of attributes to retrieve from the object's attribute database.

Return Values:

A NULL terminated list of the string representations of the attributes retrieved.

Error Return:

NULL

User Responsibility:

The user is responsible for "free"ing each returned attribute in the list as well as for "free"ing the list itself.

char**

RetrieveAllAttributes(*LegionIdType IdType, char *ObjName*);

Retrieve all of the attributes from an object's attribute database.

Parameters:

IdType =	Flag indicating whether parameter is a LOID or a context path name.
ObjName =	The context path name or the LOID of the object to which the attribute will be added.

Return Values:

A NULL terminated list of the string representations of the attributes retrieved.

Error Return:

NULL

User Responsibility:

The user is responsible for "free"ing each returned attribute in the list as well as for "free"ing the list itself.

8.6 Interfaces to Legion objects

The following function deals with the interfaces to Legion objects.

char**

GetInterface(*LegionIdType, char *ObjName*);

Retrieve all of the method signatures for a Legion object.

Parameters:

IdType =	Flag indicating whether parameter is a LOID or a context path name.
ObjName =	The context path name or the LOID of the object to retrieve the interface for.

Return Values:

A NULL terminated list of the string representations of the method's which are currently part of the indicated object's interface.

Error Return:

NULL

User Responsibility:

The user is responsible for "free"ing each returned signature in the list as well as for "free"ing the list itself.

8.7 Legion C++ type manipulation

The following section contains functions which allow for manipulation of typedefs which are normally C++ classes.

```

void
Show_CLegionHostReservation(
    CLegionHostReservation, FILE*);
void
Show_CLegionHostReservationRecord(
    CLegionHostReservationRecord, FILE*);
void
Show_CLegionVaultReservation(
    CLegionVaultReservation, FILE*);
void
Show_CLegionVaultReservationRecord(
    CLegionVaultReservationRecord, FILE*);
void
Show_CLegionReservation(CLegionReservation, FILE*);
void
Show_CLegionBinding(CLegionBinding, FILE*);
void
Show_CLegionHostObjectStatus(
    CLegionHostObjectStatus, FILE*);
void
Show_CLegionOPRAddress(CLegionOPRAddress, FILE*);
void
Show_CUVaL_InstanceRecord(CUVaL_InstanceRecord, FILE*);
void
Show_CInstancePlacementInfo(
    CInstancePlacementInfo, FILE*);
void
Show_CLegionCollectionData(CLegionCollectionData, FILE*);
    Call the C++ show function on the given C++ class (named by
    Show_XXX).

```

Parameters:

All show functions take two parameters, the type to show and the stream to "show" the output on.

Return Values:

NONE

User Responsibility: NONE

```

void
DestroyHostReservation(CLegionHostReservation);
void
DestroyHostReservationRecord(
    CLegionHostReservationRecord);
void
DestroyVaultReservation(CLegionVaultReservation);
void
DestroyVaultReservationRecord(
    CLegionVaultReservationRecord);
void
DestroyReservation(CLegionReservation);
void
DestroyLegionBinding(CLegionBinding);
void
DestroyHostObjectStatus(CLegionHostObjectStatus);
void
DestroyLegionOPRAddress(CLegionOPRAddress);
void
DestroyUVaL_InstanceRecord(CUVaL_InstanceRecord);
void
DestroyInstancePlacementInfo(CInstancePlacementInfo);
void
DestroyLegionCollectionData(CLegionCollectionData);

```

All of the complicated C++ types that are indirectly represented here as typedefs to ints must be explicitly destroyed so that their destructors may be called. To destroy an object of type xxx, call the `Destroyxxx( )` function on its C typedef.

Parameters:

All destroy functions take one parameter, the variable of the given type to destroy.

Return Values:

NONE

User Responsibility: NONE

```

int
HostReservationRecord_GetStatus(
    CLegionHostReservationRecord);
char*
HostReservationRecord_GetObjLoid(
    CLegionHostReservationRecord);
int
VaultReservationRecord_GetStatus(
    CLegionVaultReservationRecord);
char*
VaultReservationRecord_GetObjLoid(
    CLegionVaultReservationRecord);

```

```

CLegionHostReservation
LegionReservation_GetHostReservation(
    CLegionReservation);
CLegionVaultReservation
LegionReservation_GetVaultReservation(
    CLegionReservation);
CLegionHostReservation
CreateLegionHostReservation(LegionIdType, char *HostName,
    int identity, int flags);
CLegionVaultReservation
CreateLegionVaultReservation(LegionIdType,
    char *VaultName, int identity);
CLegionReservation
CreateReservation(CLegionHostReservation,
    CLegionVaultReservation);
char*
InstancePlacementInfo_GetInstance(
    CInstancePlacementInfo);
char*
InstancePlacementInfo_GetHost(CInstancePlacementInfo);
char*
InstancePlacementInfo_GetImpl(CInstancePlacementInfo);
char*
InstancePlacementInfo_GetVault(CInstancePlacementInfo);
CLegionReservation
InstancePlacementInfo_GetRSVN(CInstancePlacementInfo);
int
InstancePlacementInfo_GetArch(CInstancePlacementInfo);
CInstancePlacementInfo
CreateInstancePlacementInfo(char *InstanceLOID,
    LegionIdType, char *HostName,
    LegionIdType, char *ImplName,
    LegionIdType, char *VaultName,
    CLegionReservation, int Arch);
char*
HostObjectStatus_GetLoid(CLegionHostObjectStatus);
char*
HostObjectStatus_GetOwner(CLegionHostObjectStatus);
int
HostObjectStatus_GetActivationID(
    CLegionHostObjectStatus);
int
HostObjectStatus_GetStatus(CLegionHostObjectStatus);
int
HostObjectStatus_GetErrorCode(CLegionHostObjectStatus);
CLegionBinding
HostObjectStatus_GetBinding(CLegionHostObjectStatus);
char*
UVaL_InstanceRecord_GetLoid(CUVaL_InstanceRecord);

```

```

char*
UVaL_InstanceRecord_GetOwner(CUVaL_InstanceRecord);
int
UVaL_InstanceRecord_GetStatus(CUVaL_InstanceRecord);
char
UVaL_InstanceRecord_GetCommandLineObj(
    CUVaL_InstanceRecord);
unsigned long
UVaL_InstanceRecord_GetLastTransitionTime(
    CUVaL_InstanceRecord);
int
UVaL_InstanceRecord_GetActivationId(
    CUVaL_InstanceRecord);
char*
UVaL_InstanceRecord_GetHost(CUVaL_InstanceRecord);
char*
UVaL_InstanceRecord_GetVault(CUVaL_InstanceRecord);
int
UVaL_InstanceRecord_GetTransferringOpr(
    CUVaL_InstanceRecord);
char*
UVaL_InstanceRecord_GetTransferringToVault(
    CUVaL_InstanceRecord);
int
UVaL_InstanceRecord_GetTesting(CUVaL_InstanceRecord);
unsigned long
UVaL_InstanceRecord_GetStartTestTime(
    CUVaL_InstanceRecord);
unsigned long
UVaL_InstanceRecord_GetXferTime(CUVaL_InstanceRecord);
unsigned int
UVaL_InstanceRecord_GetHostTries(CUVaL_InstanceRecord);
char*
LegionCollectionData_GetLOID(CLegionCollectionData);
long
LegionCollectionData_GetLastUpdateTime(
    CLegionCollectionData);
/* These resource descriptions are in the form of a          /*
/* null terminated list of UVaL_ObjAttributes                /*
/* (in string form)                                          /*

```

```
char**
```

```
LegionCollectionData_GetResourceDesc(
    CLegionCollectionData);
```

All of the complicated C++ types that are indirectly represented here as typedefs to ints have internal members which are useful to access. For that purpose, a number of "accessor" functions are provided. All function names are of the form <C++ TypeName>_Get<C++ Field>.

Parameters:

All "accessor" functions take one parameter, the variable of from which the user wishes to access a member.

Return Values:

All "accessor" functions return one of three things:

- A basic integral type
- Another C typedef for a C++ class
- a char* which is the LOID of the value requested

User Responsibility:

When a C typedef for a C++ class is returned, the user is responsible for Destroying that type when done. When a char* is returned, the user is responsible for "free"ing the dynamically created memory.

8.8 Legion host object

The following section contains functions which are used to make calls directly on host objects.

CLegionHostReservation

MakeReservationFromHost(*LegionIdType HIdType*,
*char *HostName*,
LegionIdType VIdType,
*char *ReqVltName*);

Ask a host to make a reservation for either an instantiation, or an activation.

Parameters

HIdType	A flag indicating whether HostName is a context path name or a LOID.
HostName=	The name of the host with which the reservation should be made.
VIdType =	A flag indicating whether ReqVltName is a context path name or a LOID.
ReqVltName =	The name of the vault to be paired up with this host reservation.

Return Values:

The CLegionHostReservation that represents the HostReservation just made.

Error Return:

0

User Responsibility:

User is responsible for destroying the host reservation when done.

CLegionHostReservationRecord

CheckHostReservation(*LegionIdType HIdType, char *HostName, CLegionHostReservation HstRes*);

Check on the status of a host reservation previously made.

Parameters:

HIdType =	A flag indicating whether HostName is a context path name or a LOID.
HostName =	The name of the host that will check the reservation.
HstRes =	The previously made host reservation.

Return Values:

A CLegionHostReservationRecord containing the status of the reservation.

Error Return:

0

User Responsibility:

User is responsible for destroying the host reservation record when done.

int

CancelHostReservation(*LegionIdType HIdType, char *HostName, CLegionHostReservation HstRes*);

Cancel a previously made host reservation.

Parameters:

HIdType =	A flag indicating whether HostName is a context path name or a LOID.
HostName=	The name of the host that will cancel the reservation.
HstRes =	The previously made host reservation.

Return Values:

1 on Success
0 on Failure

User Responsibility: None

int

GetImplArchFromHost(*LegionIdType HIdType, char *HostName*);

Retrieve the implementation architecture for a given host.

Parameters:

HIdType =	A flag indicating whether HostName is a context path name or a LOID.
HostName=	The name of the host to get the arch from.

Return Values:

The architecture of the given host.

Error Return:

-1

User Responsibility: None

int

KillObject(*LegionIdType HIdType, char *HostName*
CLegionBinding binding, int actId);

Kill an object running on a host.

Parameters:

HIdType =	A flag indicating whether HostName is a context path name or a LOID.
HostName =	The name of the host to kill the object on.
binding =	The binding to the object that you wish to kill.
actId =	The host's activation id for the object to kill.

Return Values:

1 on Success
0 on Failure

User Responsibility: None

int

HostDeactivateObject(*LegionIdType HIdType, char *HostName,*
CLegionBinding binding, int actId);

Ask the host to deactivate an object it is running.

Parameters:

HIdType =	A flag indicating whether HostName is a context path name or a LOID.
HostName=	The name of the host to deactivate the object on.
binding =	The binding to the object that you wish to deactivate.
actId =	The host's activation id for the object to deactivate.

Return Values:

1 on Success
0 on Failure

User Responsibility: None

CLegionHostObjectStatus*

ListObjects(*LegionIdType HIdType, char *HostName*);

Ask the host to return a list of objects that it is currently managing.

Parameters:

HIdType = A flag indicating whether HostName is a
 context path name or a LOID.
HostName = The name of the host to acquire the list
 from.

Return Values:

A NULL terminated list of CLegionHostObjectStatus's. Each array element represents the status of one object currently being managed by a host.

Error Return:

NULL

User Responsibility:

The user is responsible for destroying each CLegionHostObjectStatus in the list, as well as for "free"ing the list itself.

char**

GetCompatibleVaults(*LegionIdType HIdType, char *HostName*);

Ask the host to return a list of vaults that are compatible with it.

Parameters:

HIdType = A flag indicating whether HostName is a
 context path name or a LOID.
HostName= The name of the host to check.

Return Values:

A NULL terminated list of LOIDs. Each LOID represents one vault that is compatible with the given host.

Error Return:

NULL

User Responsibility:

The user is responsible for "free"ing each LOID, as well as for "free"ing the list itself.

int

VaultOK(*LegionIdType HIdType, char *HostName,*
*LegionIdType VIdType, char *VaultName*);

Ask the host if a certain vault is compatible with it or not.

Parameters:

HIdType =	A flag indicating whether HostName is a context path name or a LOID.
HostName=	The name of the host to check the vault with.
VIdType =	A flag indicating whether VaultName is a context path name or a LOID.
VaultName=	The name of the vault to check for compatibility with.

Return Values:

0 if the vault is not OK.

1 if the vault is OK.

Error Return:

-1

User Responsibility: None

int

GetObjectStatus(*LegionIdType HIdType, char *HostName, CLegionBinding Binding*);

Ask the host for the status of one of its objects.

Parameters:

HIdType =	A flag indicating whether HostName is a context path name or a LOID.
HostName=	The name of the host to check the vault with.
Binding =	The object in question's binding.

Return Values:

The Object's status.

User Responsibility: None

char*

GetImplementationCache(*LegionIdType HIdType, char *HostName*);

Ask the host for the LOID of its ImplementationCacheObject.

Parameters:

HIdType =	A flag indicating whether HostName is a context path name or a LOID.
HostName=	The name of the host to check.

Return Values:

The LOID of the host object's implementation cache.

Error Return:
NULL

User Responsibility:
The user is responsible for “free”ing the memory returned by this function.

char*

ChangeObjectOwner(*LegionIdType HIdType, char *HostName, LegionIdType OOldType, char *OrigOwnerNm, LegionIdType NOldType, char *NewOwnerNm*);

Change the owner of a certain object that the host is running.

Parameters:

HIdType = A flag indicating whether HostName is a context path name or a LOID.
 HostName = The name of the host to check the vault with.
 OOldType = A flag indicating whether OrigOwnerNm is a context path name or a LOID.
 OrigOwnerNm = The name of the original owner of the object.
 NOldType = A flag indicating whether NewOwnerNm is a context path name or a LOID.
 NewOwnerNm = The name of the new owner of the object.

Return Values:
The LOID of the object’s owner.

Error Return:
NULL

User Responsibility:
The user is responsible for “free”ing the memory returned by this function.

8.9 Vault object

The following section contains functions which are used to make calls directly on host objects.

C LegionVaultReservation

MakeReservationFromVault(*LegionIdType VIdType, char *VaultName, LegionIdType HIdType, char *ReqHostNm*);

Ask a vault to make a reservation for either an instantiation or an activation.

Parameters:

VldType = A flag indicating whether VaultName is a context path name or a LOID.
 VaultName = The name of the vault that will make the reservation.
 HldType = A flag indicating whether ReqHostName is a context path name or a LOID.
 ReqHostName = The name of the host to be paired up with this vault reservation.

Return Values:

The CLegionVaultReservation that represents the VaultReservation just made.

Error Return:

0

User Responsibility:

User is responsible for destroying the vault reservation when done.

CLegionVaultReservationRecord

CheckVaultReservation(*LegionIdType VldType,*
*char *VaultName, CLegionVaultReservation VtRes*);
 Check on the status of a vault reservation previously made.

Parameters:

HldType = A flag indicating whether VaultName is a context path name or a LOID.
 VaultName = The name of the vault that will check the reservation.
 VtRes = The Vault reservation previously made.

Return Values:

A CLegionVaultReservationRecord containing the status of the reservation.

Error Return:

0

User Responsibility:

User is responsible for destroying the vault reservation record when done.

int

CancelVaultReservation(*LegionIdType VldType,*
*char *VaultName, CLegionVaultReservation VtRes*);
 Cancel a previously made vault reservation.

Parameters:

HldType = A flag indicating whether VaultName is a context path name or a LOID.

VaultName= The name of the vault that will cancel the reservation.
VtRes = The vault reservation previously made.

Return Values:
1 on Success
0 on Failure

User Responsibility: None

int

HostOK(*LegionIdType VldType, char *VaultName,*
*LegionIdType HldType, char *HostName*);

Ask the vault whether or not a certain host is compatible with it.

Parameters:

VldType = A flag indicating whether VaultName is a context path name or a LOID.
VaultName= The name of the vault that will check the host.
HldType = A flag indicating whether HostName is a context path name or a LOID.
HostName= The name of the host to check for compatibility.

Return Values:
0 if the host is not OK.
1 if the host is OK.

Error Return:
-1

User Responsibility: None

char**

GetCompatibleHosts(*LegionIdType VldType, char *VaultName*);

Ask the vault to return a list of hosts that are compatible with it.

Parameters:

VldType = A flag indicating whether VaultName is a context path name or a LOID.
VaultName= The name of the vault to check.

Return Values:
A NULL terminated list of LOIDs. Each LOID represents one host that is compatible with the given vault.

Error Return:
NULL

User Responsibility:

The user is responsible for “free”ing each LOID, as well as for “free”ing the list itself.

int

TransferOPRsAndDestroySelf(*LegionIdType VldType*,
*char *VaultName*);

Ask the vault to transfer all of its OPRs to another vault, and then to die.

Parameters:

VldType = A flag indicating whether VaultName is a
 context path name or a LOID.
VaultName= The name of the vault to talk to.

Return Values:

1 on Success
0 on Failure

User Responsibility: None

char*

ChangeOPROwner(*LegionIdType VldType*, *char *VaultName*,
LegionIdType OOldType, *char *OrigOwnerNm*,
LegionIdType NOldType, *char *NewOwnerNm*);

Change the owner of a certain OPR that the vault is managing.

Parameters:

VldType = A flag indicating whether VaultName is a
 context path name or a LOID.
VaultName = The name of the vault to talk with.
OOldType = A flag indicating whether OrigOwnerNm is
 a context path name or a LOID.

OrigOwnerNm = The name of the original owner of the OPR.

NOldType = A flag indicating whether NewOwnerNm is
 a context path name or a LOID.

NewOwnerNm = The name of the new owner of the OPR.

Return Values:

The LOID of the OPR’s owner.

Error Return:

NULL

User Responsibility:

The user is responsible for “free”ing the memory returned by this function.

8.10 Class object

The following section contains a set of functions which interface directly with Class objects.

int

AddImplementation(*LegionIdType CIdType, char *ClassName, LegionIdType IIdType, char *ImplName, int Arch*);

Adds a new implementation of a class binary to the given class.

Parameters:

CIdType =	A flag indicating whether ClassName is a LOID or a context path.
ClassName =	The name of the class to add the implementation to.
IIdType =	A flag indicating whether ImplName is a LOID or a context path.
ImplName =	The name of an implementation object to add.
Arch =	The architecture for which the implementation is valid.

Return Values:

1 on Success
0 on Failure

User Responsibility: None

int

DeleteInstance(*LegionIdType CIdType, char *ClassName, LegionIdType IIdType, char *InstanceNm*);

Asks a class to delete one of its instances.

Parameters:

CIdType =	A flag indicating whether ClassName is a LOID or a context path.
ClassName =	The name of the class to delete the instance from.
IIdType =	A flag indicating whether InstanceNm is a LOID or a context path.
InstanceNm=	The name of an object instance to delete.

Return Values:

1 on Success
0 on Failure

User Responsibility: None

int

ActivateInstance(*LegionIdType CldType, char *ClassName, CInstancePlacementInfo CInfo*);

Asks a class to activate one of its instances.

Parameters:

CldType = A flag indicating whether ClassName is a LOID or a context path.

ClassName = The name of the class to activate the instance for.

CInfo = A CInstancePlacementInfo variable indicating which instance to activate and where.

Return Values:

1 on Success

0 on Failure

User Responsibility: None

int

DeactivateInstance(*LegionIdType CldType, char *ClassName, LegionIdType OldType, char *ObjName*);

Asks a class to deactivate one of its instances.

Parameters:

CldType = A flag indicating whether ClassName is a LOID or a context path.

ClassName = The name of the class to deactivate the instance for.

OldType = A flag indicating whether ObjName is a LOID or a context path.

ObjName = A LOID or context path indicating which object to deactivate.

Return Values:

1 on Success

0 on Failure

User Responsibility: None

int

ForcedDeactivateInstance(
*LegionIdType CldType, char *ClassName,*
*LegionIdType OldType, char *ObjName*);

Asks a class to deactivate one of its instances. If the object is in the going-up state or in the going-down state, this call will fail.

Parameters:

CldType = A flag indicating whether ClassName is a LOID or a context path.

ClassName = The name of the class whose instance is to be deactivated.
 OldType = A flag indicating whether ObjName is a LOID or a context path.
 ObjName = A LOID or context path indicating which object to deactivate.

Return Values:
 1 on Success
 0 on Failure

User Responsibility:
 None

CUVaL_InstanceRecord*

GetInstanceList(*LegionIdType CldType, char *ClassName*);

Asks a class to return a list of all of the instances that it is currently managing.

Parameters:

CldType = A flag indicating whether ClassName is a LOID or a context path.
 ClassName = The name of the class to retrieve the list from.

Return Values:

A NULL terminated list of CUVaL_InstanceRecords where each element in the array is the UVaL_InstanceRecord for one of the objects that the class is managing.

Error Return:
 NULL

User Responsibility:

The user is responsible for destroying each CUVaL_InstanceRecord in the list, as well as for “free”ing the list itself.

8.11 Legion implicit parameters

The following section contains the functions which allow a program to manipulate its own method and message implicit parameters.

int

MethodImplicitParm_InsertLOID(*char *ParmName, char *Loid*);

Insert a LOID implicit parameter into the Implicit method parameters

Parameters:

ParmName = The name of the implicit parameter to which the action is being applied.
Loid = The String representation of the LOID to add.

Return Values:
0 on Failure
1 on Success

User Responsibility: None

int

MethodImplicitParm_InsertInt(*char *ParmName, int Value*);

Insert an int implicit parameter into the Implicit method parameters

Parameters:

ParmName = The name of the implicit parameter to which the action is being applied.
Value = The integer value to be placed into the parameters.

Return Values:
0 on Failure
1 on Success

User Responsibility: None

int

MethodImplicitParm_InsertString(*char *ParmName, char *Str*);

Insert a new string implicit parameter into the implicit method parameter database.

Parameters:

ParmName = The name of the implicit parameter to which the action is being applied.
Str = The new string to add to the database.

Return Values:
0 on Failure
1 on Success

User Responsibility: None

int

MethodImplicitParm_Remove(*char *ParmName*);

Remove an implicit parameter from the implicit method parameters database.

Parameters:

ParmName = The name of the implicit parameter to which the action is being applied.

Return Values:

0 on Failure

1 on Success

User Responsibility: None

char*

MethodImplicitParm_FindLOID(char *ParmName);

Find an implicit parameter in the implicit method parameters database (this parameter must be a LOID parameter).

Parameters:

ParmName = The name of the implicit parameter to which the action is being applied.

Return Values:

The string representation of the LOID that was found.

Error Return:

NULL

User Responsibility:

The user is responsible for "free"ing the returned LOID when done.

int

MethodImplicitParm_FindInt(char *ParmName);

Find an implicit parameter in the implicit method parameters database (this parm must be an int parameter).

Parameters:

ParmName = The name of the implicit parameter to which the action is being applied.

Return Values:

The integer parameter that was retrieved from the database.

Error Return:

Need to check LegionErrno for this one.

User Responsibility: None

char*

MethodImplicitParm_FindString(char *ParmName);

Retrieve a string parameter from the implicit method parameter database.

Parameters:

ParmName = The name of the implicit parameter to which the action is being applied.

Return Values:

The string parameter retrieved from the implicit method database.

Error Return:

NULL

User Responsibility:

The user is responsible for “free”ing the memory returned from this function call.

int

MethodImplicitParm_ReplaceLOID(

*char *ParmName, char *Loid*);

Replace an implicit method parameter in the database with another (LOID) parameter.

Parameters:

ParmName = The name of the implicit parameter to which the action is being applied.

Loid = The LOID parameter to replace <ParmName> with.

Return Values:

0 on Failure

1 on Success

User Responsibility: None

int

MethodImplicitParm_ReplaceInt(char *ParmName, int Value);

Replace an implicit method parameter in the database with another (int) parameter.

Parameters:

ParmName = The name of the implicit parameter to which the action is being applied.

Value = The value of the integer parameter to replace <ParmName> with.

Return Values:

0 on Failure

1 on Success

User Responsibility: None

int

MethodImplicitParm_ReplaceString

*char *ParmName, char *Str*;

Replace an implicit method parameter in the database with another (string) parameter.

Parameters:

ParmName = The name of the implicit parameter to which the action is being applied.

Str = The string parameter which the user wishes to replace <ParmName> with.

Return Values:

0 on Failure

1 on Success

User Responsibility: None

int

MessageImplicitParm_InsertLOID(*char *ParmName, char *Loid*);

Insert a LOID implicit parameter into the implicit message parameters

Parameters:

ParmName = The name of the implicit parameter to which the action is being applied.

Loid = The string representation of the LOID to be added.

Return Values:

0 on Failure

1 on Success

User Responsibility:

None

int

MessageImplicitParm_InsertInt(*char *ParmName, int Value*);

Insert an int implicit parameter into the implicit message parameters.

Parameters:

ParmName = The name of the implicit parameter to which the action is being applied.

Value = The integer value to be placed into the parameters.

Return Values:

0 on Failure

1 on Success

User Responsibility: None

int

MessageImplicitParm_InsertString(*char *ParmName, char *Str*);

Insert a new String Implicit Parameter into the implicit message parameter database.

Parameters:

ParmName = The name of the implicit parameter to which the action is being applied.

Str = The new string to add to the database.

Return Values:

0 on Failure

1 on Success

User Responsibility: None

int

MessageImplicitParm_Remove(*char *ParmName*);

Remove an implicit parameter from the implicit message parameters database.

Parameters:

ParmName = The name of the implicit parameter to which the action is being applied.

Return Values:

0 on Failure

1 on Success

User Responsibility: None

char*

MessageImplicitParm_FindLOID(*char *ParmName*);

Find an implicit parameter in the implicit message parameters database (this parm must be a LOID parameter).

Parameters:

ParmName = The name of the implicit parameter to which the action is being applied.

Return Values:

The string representation of the LOID that was found.

Error Return:

NULL

User Responsibility:

The user is responsible for “free”ing the returned LOID when done.

int

MessageImplicitParm_FindInt(char *ParmName);

Find an implicit parameter in the implicit message parameters database (this parm must be an int parameter).

Parameters:

ParmName = The name of the implicit parameter to which the action is being applied.

Return Values:

The integer parameter that was retrieved from the database.

Error Return:

Need to check LegionErrno for this one.

User Responsibility: None

char*

MessageImplicitParm_FindString(char *ParmName);

Retrieve a string parameter from the implicit message parameter database.

Parameters:

ParmName = The name of the implicit parameter to which the action is being applied.

Return Values:

The string parameter retrieved from the implicit message database.

Error Return:

NULL

User Responsibility:

The user is responsible for “free”ing the memory returned from this function call.

int

MessageImplicitParm_ReplaceLOID(

*char *ParmName, char *Loid*);

Replace an implicit message parameter in the database with another (LOID) parameter.

Parameters:

ParmName = The name of the implicit parameter to which the action is being applied.

Loid = The Loid parameter to replace
 <ParmName> with.

Return Values:
0 on Failure
1 on Success

User Responsibility: None

int

MessageImplicitParm_ReplaceInt(*char *ParmName, int Value*);
Replace an implicit message parameter in the database with
another (int) parameter.

Parameters:
ParmName = The name of the implicit parameter to
 which the action is being applied.
Value = The value of the integer parameter to
 replace <ParmName> with.

Return Values:
0 on Failure
1 on Success

User Responsibility: None

int

MessageImplicitParm_ReplaceString(*char *ParmName,*
 *char *Str*);
Replace an implicit message parameter in the database with
another (string) parameter.

Parameters:
ParmName = The name of the implicit parameter to
 which the action is being applied.
Str = The string parameter which the user
 wishes to replace <ParmName> with.

Return Values:
0 on Failure
1 on Success

User Responsibility: None

8.12 Collections

The following section contains the functions which allow a program to
manipulate, update, and query Legion collection objects.

int

JoinCollection(

LegionIdType CollectionIdType, char CollectionName,*

*LegionIdType JoinerIdType, char *JoinerName);*

Join an object into a collection

Parameters:

CollectionIDType = A flag indicating whether
CollectionName is a LOID or a
context name.

CollectionName = Either the LOID or the context path of
the collection object in question.

JoinerIDType = A flag indicating whether JoinerName
is a LOID or a context path.

JoinerName = Either a LOID or a context path for the
object being added to the collection.

Return Values:

0 on Failure

1 on Success

User Responsibility: None

int

JoinCollection_wData(

LegionIdType CollectionIdType, char CollectionName,*

*LegionIdType JoinerIdType, char *JoinerName,*

*char **Attributes);*

Join an object into a collection and give init data.

Parameters:

CollectionIDType = A flag indicating whether
CollectionName is a LOID or a
context name.

CollectionName = Either the LOID or the context path of
the collection object in question.

JoinerIDType = A flag indicating whether JoinerName
is a LOID or a context path.

JoinerName = Either a LOID or a context path for the
object being added to the collection.

Attributes = NULL terminated array of object
attributes.

Return Values:

0 on Failure

1 on Success

User Responsibility: None

int

LeaveCollection(
 LegionIdType CollectionIdType, *char** CollectionName,
 LegionIdType LeaverIdType, *char **LeaverName);

Indicate that an object is leaving a Collection.

Parameters:

CollectionIDType =	A flag indicating whether CollectionName is a LOID or a context name.
CollectionName =	Either the LOID or the context path of the collection object in question.
LeaverIDType =	A flag indicating whether LeaverName is a LOID or a context path.
LeaverName =	Either a LOID or a context path for the object leaving the collection.

Return Values:

0 on Failure
1 on Success

User Responsibility: None

int

UpdateCollectionEntry(
 LegionIdType CollectionIdType, *char** CollectionName,
 LegionIdType ConstituentIdType, *char **ConstituentName,
 *char ***Attributes);

Update a collection entry's data.

Parameters:

CollectionIDType =	A flag to show if CollectionName is a LOID or a context name.
CollectionName =	Either the LOID or the context path of the collection object in question.
ConstituentIDType =	A flag to show if ConstituentName is a LOID or a context path.
ConstituentName =	Either a LOID or a context path for the object whose collection data is being updated.
Attributes =	NULL terminated array of object attributes.

Return Values:

0 on Failure
1 on Success

User Responsibility: None

CLegionCollectionData*

QueryCollection(

```
LegionIdType CollectionIdType, char *CollectionName,
char *QueryString);
```

Retrieve a list of queried information from the collection.

Parameters:

CollectionIDType = A flag to show if CollectionName is a LOID or a context name.

CollectionName = Either the LOID or the context path of the collection object in question.

QueryString = A MESSIAHS-style query string

Return Values:

A NULL terminated list of CollectionData entries

Error Returns:

Returns NULL on failure

User Responsibility:

User is responsible for destroying the individual elements returned, as well as for “free”ing up the array.

```
#ifndef_cplusplus
```

}

```
#endif
```

```
#endif
```

9.0 Error messages

The CError.h file contains the declarations for the error reporting mechanism used in the C Library interface to the Legion system. Below are the identifiers and their codes.

#define LEGION_NO_ERROR	0
#define LEGION_NOT_A_LOID	1
#define LEGION_NOT_IN_CONTEXT_SPACE	2
#define LEGION_UNKNOWN_ERROR	3
#define LEGION_CONTEXT_SPACE_INVALID	4
#define LEGION_ROOT_CONTEXT_ALREADY_EXISTS	5
#define LEGION_CONTEXT_ALREADY_EXISTS	6
#define LEGION_NO_CONTEXT_CLASS	7
#define LEGION_CONTEXT_CREATE_FAILED	8
#define LEGION_UNABLE_TO_ADD_PATH	9
#define LEGION_METHOD_TIMED_OUT	10
#define LEGION_BAD_ATTRIBUTE_FORMAT	11
#define LEGION_UNABLE_TO_GET_INTERFACE	12
#define LEGION_PARM_ALREADY_EXISTS	13
#define LEGION_PARM_DOES_NOT_EXIST	14
#define LEGION_PARM_WRONG_TYPE	15
#define LEGION_PARM_NO_DATA	16
#define LEGION_PARM_UNABLE_TO_REPLACE	17
#define LEGION_NOT_A_CLASS	18
#define LEGION_NOT_A_HOST	19
#define LEGION_NOT_A_VAULT	20
#define LEGION_NOT_AN_IMPL	21
#define LOID_NOT_EXPECTED	22
#define LOID_EXPECTED	23
#define LEGION_PARM_TYPE_UNKNOWN	24
#define LEGION_OUTCALL_FAILED	25
#define LEGION_OUTCALL_PERMISSION_DENIED	26
#define LEGION_CLASS_NOT_FOUND	27
#define LEGION_BAD_ARGUMENT	28
#define LEGION_OBJECT_TYPE	29
#define LEGION_NO_CURRENT_CONTEXT	30
#define LEGION_OBJECT_NOT_FOUND	31
#define LEGION_INTERNAL_ERROR	32
#define LEGION_EMPTY_LOID	33

Getting help

Please contact us at one of the addresses listed below if you have trouble with the system. Please be sure to include any relevant information about the state of your system before and during the problem.

We would greatly appreciate hearing from you whenever you find errors or bugs in Legion, so that we can avoid similar problems in future releases.

E-mail

For bug reports, help, and general Legion information, please send an e-mail message to legion-help@virginia.edu

Contact Information

Legion Group
Department of Computer Science
School of Engineering & Applied Science
University of Virginia
151 Engineer's Way, P.O. Box 400740
Charlottesville, VA 22904-4740
<http://legion.virginia.edu/>
fax: 434-982-2214

On-Line Help

A variety of technical notes, reports, and on-line tutorials are available on the Legion web site, at <http://legion.virginia.edu/>.

Index

A

accept_member_functions()	122
accounting logs	57
architectures available in Legion	103, 108

B

bootstrap host	120
----------------	-----

C

cat	26
CError.h file	115, 141, 202
CLegionLib.h file	141
commands	
documentation	6
legion_2drm	37
legion_activate_instances	26
legion_activate_object	14
legion_add_acl	62
legion_add_class_mapping	22
legion_add_host_account	38
legion_add_implicit_params	62
legion_add_sub_collection	45
legion_allow_activation	26
legion_bfs	73
legion_cat	26
legion_cd	26
legion_change_acl	62
legion_change_owner	63
legion_change_permissions	64
legion_class_host_list	45
legion_class_vault_list	7, 46
legion_classof	57
legion_combine_domains	22
legion_config_scheduler	47
legion_configure_profile	7
legion_context_add	27
legion_context_create	27
legion_copy_class	14
legion_cp	27
legion_create_class	38
legion_create_implementation	23
legion_create_object	15
legion_create_object_r	15
legion_create_stat_tree	57
legion_create_user	65
legion_create_user_object	66
legion_deactivate_instances	29
legion_deactivate_object	16
legion_destroy_host	39
legion_destroy_instances	29

legion_destroy_object	17
legion_destroy_vault	39
legion_direct_output	29
legion_export_dir	30
legion_export_dir_quit	31
legion_export_dir_rehash	31
legion_exports_interface	8
legion_ft_initialize	80
legion_FTPd	9
legion_generate_domain_cookie	24
legion_get_accounting_data	57
legion_get_acl	67
legion_get_host	31
legion_get_implicit_params	67
legion_get_interface	9
legion_get_vault	31
legion_host_stats	58
legion_host_vault_list	47
legion_init_Apps	67
legion_init_arch	24
legion_init_Extra	67
legion_init_HPC	67
legion_init_security	67
legion_initialize	39
legion_instance_host_list	48
legion_instance_vault_list	49
legion_join_collection	49
legion_leave_collection	50
legion_link	80, 81
legion_list_attributes	10
legion_list_domains	25
legion_list_host_accounts	40
legion_list_implementations	17
legion_list_instances	17
legion_list_invocations	10
legion_list_names	31
legion_list_objects	58
legion_list_oprs	50
legion_list_sub_collections	50
legion_In	32
legion_login	68
legion_logout	68
legion_Is	32
legion_make	73
legion_make_multi	74
legion_make_schedule	50
legion_make_setup_script	40
legion_manage_job	82
legion_manage_queue	83
legion_mkdir	33
legion_modify_parameters	69
legion_mpi_debug	83
legion_mpi_probe	84
legion_mpi_register	85

legion_mpi_run	85
legion_mplc	75
legion_mplc_reg_impl	75
legion_mv	34
legion_native_mpi_config_host	90, 91
legion_native_mpi_init	91
legion_native_mpi_register	91
legion_native_mpi_run	92
legion_nq	93
legion_object_info	10
legion_output_state	75
legion_passwd	69
legion_ping	9
legion_print_config	40
legion_print_domain_cookie	25
legion_probe_run	96
legion_pvm_register	98
legion_pwd	34
legion_query_collection	51
legion_record	76
legion_refresh_local_cache	18
legion_register_program	99
legion_register_runnable	99
legion_remove_host_account	41
legion_remove_sub_collection	53
legion_replay	77
legion_rm	34
legion_run	100
legion_run_multi	104
legion_set_acl	70
legion_set_backup_vaults	7, 18
legion_set_binding_agent	19
legion_set_context	35
legion_set_default_placement	53
legion_set_host	19
legion_set_implicit_params	70
legion_set_message_security	70
legion_set_scheduler	54
legion_set_scheduler_policy	54
legion_set_tty	35
legion_set_varch	55
legion_set_vault	20
legion_set_vrun	55
legion_set_worm	12
legion_setup_state	41
legion_show_acl	71
legion_shutdown	41
legion_shutdown_class	41
legion_skcc_set_class_vaults	7, 11
legion_skcc_set_defaults	7, 11
legion_starthost	42
legion_startup	43, 119
legion_startvault	43
legion_stateless_add_workers	78

legion_stateless_configure	78
legion_stateless_remove_workers	79
legion_synch_vaults	21
legion_tty	35
legion_tty_off	35
legion_tty_redirect	36
legion_tty_unredirect	36
legion_tty_watch	36
legion_unset_worm	12
legion_update_accounting_db	58
legion_update_attributes	12
legion_update_stat_tree	59
legion_vault_host_list	55
legion_version	60
legion_wellknown_class	60
legion_whereis	60
legion_whoami	61
Constructors	132
context space	
documentation	6
cp	27
E	
enable_functions()	122
F	
fault tolerance	
changing SKCC settings	7
G	
getInterface()	9
grid library	
attributes	
AddAttribute()	168
AddAttributes()	168
GetInterface()	175
RemoveAttribute()	171
RemoveAttributes()	172
RemoveAttributeSig()	171
RemoveAttributeSigs()	172
ReplaceAttribute()	169
ReplaceAttributes()	170
ReplaceAttributeSig()	169
ReplaceAttributeSigs()	170
RetrieveAllAttributes()	175
RetrieveAttribute()	173
RetrieveAttributes()	173
RetrieveAttributeSig()	174
RetrieveAttributeSigs()	174
C++ type manipulation	
CreateInstancePlacementInfo()	178
CreateLegionHostReservation()	178
CreateLegionVaultReservation()	178
CreateReservation()	178

DestroyHostObjectStatus()	177
DestroyHostReservation()	177
DestroyHostReservationRecord()	177
DestroyInstancePlacementInfo()	177
DestroyLegionBinding()	177
DestroyLegionCollectionData	177
DestroyLegionOPRAddress()	177
DestroyReservation()	177
DestroyUVaL_InstanceRecord()	177
DestroyVaultReservation()	177
DestroyVaultReservationRecord()	177
HostObjectStatus_GetActivationID()	178
HostObjectStatus_GetBinding()	178
HostObjectStatus_GetErrorCode()	178
HostObjectStatus_GetLoid()	178
HostObjectStatus_GetOwner()	178
HostObjectStatus_GetStatus()	178
HostReservationRecord_GetObjLoid()	177
HostReservationRecord_GetStatus()	177
InstancePlacementInfo_GetArch()	178
InstancePlacementInfo_GetHost()	178
InstancePlacementInfo_GetImpl()	178
InstancePlacementInfo_GetInstance()	178
InstancePlacementInfo_GetRSVN()	178
InstancePlacementInfo_GetVault()	178
LegionCollectionData_GetLastUpdateTime	179
LegionCollectionData_GetLOID	179
LegionCollectionData_GetResourceDesc	179
LegionReservation_GetHostReservation()	178
LegionReservation_GetVaultReservation()	178
Show_CInstancePlacementInfo()	176
Show_CLegionBinding()	176
Show_CLegionCollectionData	176
Show_CLegionHostObjectStatus()	176
Show_CLegionHostReservation()	176
Show_CLegionHostReservationRecord()	176
Show_CLegionOPRAddress()	176
Show_CLegionReservation()	176
Show_CLegionVaultReservation()	176
Show_CLegionVaultReservationRecord()	176
Show_CUVaL_InstanceRecord()	176
UVaL_InstanceRecord_GetActivationId()	179
UVaL_InstanceRecord_GetCommandLineObj()	179
UVaL_InstanceRecord_GetHost()	179
UVaL_InstanceRecord_GetHostTries()	179
UVaL_InstanceRecord_GetLastTransitionTime()	179
UVaL_InstanceRecord_GetLoid()	178
UVaL_InstanceRecord_GetOwner()	179
UVaL_InstanceRecord_GetStartTestTime()	179
UVaL_InstanceRecord_GetStatus()	179
UVaL_InstanceRecord_GetTesting()	179
UVaL_InstanceRecord_GetTransferringOpr()	179
UVaL_InstanceRecord_GetTransferringToVault()	179
UVaL_InstanceRecord_GetVault()	179

UVaL_InstanceRecord_GetXferTime()	179
VaultReservationRecord_GetObjLoid()	177
VaultReservationRecord_GetStatus()	177
CError.h file	141
class objects	
ActivateInstance()	190
AddImplementation()	189
DeactivateInstance()	190
DeleteInstance()	189
ForcedDeactivateInstance()	190
GetInstanceList()	191
CLegionLib.h file	141
collections	
JoinCollection()	199
JoinCollection_wData()	199
LeaveCollection()	200
QueryCollection()	201
UpdateCollectionEntry()	200
context space	
ContextAdd()	149
ContextLookup()	147
ContextMultiLookup()	148
ContextMultiReverseLookup()	150
ContextPathAdd()	149
ContextPathLookup()	147
ContextPathMultiLookup()	148
ContextPathRemove()	150
ContextPWD()	151
ContextRemove()	149
ContextSingleReverseLookup()	150
ContextSpaceActive()	147
legion_cd()	152
legion_mkdir()	151
exceptions interface	
Legion_ExceptionCatcherDefaultEnable()	146
host objects	
CancelHostReservation()	181
ChangeObjectOwner()	185
CheckHostReservation()	181
GetCompatibleVaults()	183
GetImplArchFromHost()	181
GetImplementationCache()	184
GetObjectStatus()	184
HostDeactivateObject()	182
KillObject()	182
ListObjects()	183
MakeReservationFromHost()	180
VaultOK()	183
implicit parameters	
MessageImplicitParm_FindInt()	197
MessageImplicitParm_FindLOID()	196
MessageImplicitParm_FindString()	197
MessageImplicitParm_InsertInt()	195
MessageImplicitParm_InsertLOID()	195

MessageImplicitParm_InsertString()	196
MessageImplicitParm_Remove()	196
MessageImplicitParm_ReplaceInt()	198
MessageImplicitParm_ReplaceLOID()	197
MessageImplicitParm_ReplaceString()	198
MethodImplicitParm_FindInt()	193
MethodImplicitParm_FindLOID()	193
MethodImplicitParm_FindString()	193
MethodImplicitParm_InsertInt()	192
MethodImplicitParm_InsertLOID()	191
MethodImplicitParm_InsertString()	192
MethodImplicitParm_Remove()	192
MethodImplicitParm_ReplaceInt()	194
MethodImplicitParm_ReplaceLOID()	194
MethodImplicitParm_ReplaceString()	195
Legion Library interface	143
ClassOf()	145
GetMyLOID()	146
Legion_AcceptMethods()	144
Legion_DeleteSelf()	145
Legion_GetLegionClassBinding()	144
Legion_GetLegionClassLOID()	144
Legion_init()	143
Legion_init_command_line_class()	143
Legion_init_with_class()	143
Legion_Sleep()	145
objects	
Activate()	158
ActivateRestricted()	159
ComplxActivateRestricted()	160
ComplxCreateRestricted()	156
ComplxScheduledActivateRestricted()	161
ComplxScheduledCreateRestricted()	157
Create()	153
CreateRestricted()	154
DeactivateObject()	162
DeleteObject()	162
DestroyObject()	162
ForcedDeactivateObject()	163
GetObjectType()	163
isBindingAgent()	167
isClass()	165
isContext()	164
isFile()	164
isHost()	165
isImplementation()	166
isImplementationCache()	166
isLegionClass()	167
isMPI()	167
isPVM3()	168
isRunnable()	164
isTTY()	166
isVault()	165
ObjectActive()	152

Ping()	152
ScheduledActivate()	158
ScheduledActivateRestricted()	159
ScheduledCreate()	153
ScheduledCreateRestricted()	155
valid architectures	142
vault objects	
CancelVaultReservation()	186
ChangeOPROwner()	188
CheckVaultReservation()	186
GetCompatibleHosts()	187
HostOK()	187
MakeReservationFromVault()	185
TransferOPRsAndDestroySelf()	188

H

host object	
adding new, command-line procedure	119
adding new, rsh procedure	119
restart extras, command-line	120
shutting down extras, command-line	120
hosts	
bootstrap host	120
start-up host	120

I

internal tools	
legion_accounting_info_pusher	117
legion_init_Apps_security	117
legion_init_Extra_security	117
legion_init_HPC_security	117
legion_init_SDK	117
legion_init_SDK_security	117
legion_make_backend	117
legion_modify_schedule	117
legion_mplfront	117
legion_native_mpi_backend	118
legion_native_mpich_wrapper	118
legion_native_mpiibm_wrapper	118
legion_native_mpisgi_wrapper	118
legion_native_mpiultra_wrapper	118
legion_permit_user	118
legion_probe_object	118
legion_translate_arch	118
legion_vrun_kill	118
legion_vrun_run	118
legion_watch_process	118
invoke_method()	122

L

Legion documentation	5
legion_accounting_info_pusher	117
legion_make_backend	117
legion_modify_schedule	117

legion_mplfront	117
legion_native_mpi_backend	118
legion_native_mpich_wrapper	118
legion_native_mpiibm_wrapper	118
legion_native_mpisgi_wrapper	118
legion_native_mpiultra_wrapper	118
legion_permit_user	118
legion_probe_object	118
Legion_Simple	122
legion_translate_arch	118
legion_watch_process	118
LegionBuffer	132
associated LegionCompressor	135
associated LegionEncryptor	135
associated LegionPacker	134
associated LegionStorage	133
constructors	132
LegionBuffer()	132–133
LegionPackable	135
show()	135
LegionCompressor	135
compress()	135
decompress()	135
LegionComputationTag	140
accessor functions	140
constructors	140
pack()	140
show()	140
unpack()	140
LegionEncryptor	135
decode()	135
encrypt()	135
LegionInvocationStore	131
any_ready()	131
any_ready_for_func()	131
enable_function ()	131
get_return_value()	132
insert()	131
next_matched()	131
next_matched_for_func()	131
release_all_return_values()	132
release_return_value()	132
set_priority ()	131
LegionLOID	135
accessors	137
is_class()	137
is_empty()	137
LegionLOID()	137
LegionLOID() (protected)	136
overloaded operators	137
pack()	137
same_class_as()	137
set_field_size()	136
set_type()	136
show()	138

unpack()	137
LegionMessage	138
accessor functions	138
LegionContinuationList	140
LegionEnvironment	140
LegionEnvironmentItem	140
LegionMessage()	138
other fields	140
overloaded operators	138
pack()	138
show()	138
unpack()	138
LegionPackable	135
pack()	135
LegionPacker	134
get_ZZZ()	135
put_ZZZ()	134
LegionParameter	138
accessor functions	139
LegionParameter()	138
other constructors	139
overloaded operators	139
pack()	139
show()	139
unpack()	139
LegionParameterList	139
constructors	139
find()	139
pack()	140
set operations	139
show()	139
unpack()	140
LegionProgramGraph	130
add_constant_parameter()	130
add_invocation ()	130
add_invocation_parameter()	130
add_result_dependency()	130
execute()	131
get_value()	131
LegionStorage	133
current_byte()	134
getMetaData()	134
linearize()	134
read()	133
seek()	133
setMetaData()	134
size()	134
write()	133
LegionWorkUnit	132
get_continuation_list()	132
get_function_number()	132
get_parameter()	132
library implementation interfaces	130
In	32
Is	32

M

Mentat	75
message layer	
changing security mode	7
mkdir	27, 33
mv	34

N

naming and binding	
LegionLOID	135

P

ping()	9
pwd	34

R

rm	34
runtime library	
ex1_Simple.c	127
examples and interfaces	122
Simple.c	122
Simple.h	122
Simple.trans.c	123
Simple.trans.h	123

S

simple class	122
start a single system on two hosts	120
start-up host	120

U

user profile	7
--------------	---

V

vault object	
adding new, command-line procedure	119
adding new, rsh procedure	119
restart extras, command-line	120
shutting down extras, command-line	120